A QUALITATIVE COMPARISON OF CODING LANGUAGES USED FOR IMAGE SYNTHESIS

A Thesis

by

CHRISTOPHER STEVEN POTTER

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Ergun Akleman
Committee Members,	Ann McNamara
	John Keyser
Head of Department,	Tim McLaughlin

December 2015

Major Subject: Visualization

ABSTRACT

In this thesis four different computer programming languages, C++, Python, Processing, and Pixar's RenderMan© , were used to realize four different rendering programs. The goal was to identify the main challenges in implementation with each language and qualitatively evaluate each program once completed. A history of ray casting and rendering theory is introduced. Then, a set of "ray tracing milestones" were established so that each language can address the challenges unique to that language. These milestones are related to the image synthesis process and include preliminary preparations, direct illumination, distributed ray tracing, and indirect illumination.

After writing and reviewing with the four different computer programming languages, it was found that Processing offers the best opportunity for thoroughly implementing a rendering program because it will allow more time to be focused on rendering and ray casting theory, rather than language implementation process. It is inevitable that some learning must occur for all scripting languages with specific syntax-related challenges, but Processing's pre-packaging plug-and-play system makes the most versatile, accessible and requires a smaller learning curve than the other languages provided.

DEDICATION

I dedicate this thesis to my wife Stefanie. Without her encouragement this thesis would never have been finished.

ACKNOWLEDGEMENTS

I would like thank my wife, Stefanie, for all her support and encouragement throughout this thesis process. I would like to thank Ergun for supporting my thesis idea and guiding my development process, and Ann and John for taking the time to review my paper and challenge me during my defense. I would like to give a special thank you to Jamie, Phillip, Stefanie and Penny for proofing this paper and helping mold it into a professional document. I would like to thank my parents for their ever loving support and guidance to success throughout my life and throughout this Master's thesis, and finally the Vizlab for fostering an environment of learning and growth that has helped me immensely in my career today. Without the VizLab I would not be where I am in life today.

NOMENCLATURE

GUI	Graphical User Interface
IDE	Integrated Developing Environment
MIT	Massachusetts Institute of Technology
PIL	Python Imaging Library
RIB	RenderMan©Interface Bytestream
RSL	RenderMan©Shading Language

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Image synthesis is the process of generating images. For the purpose of this thesis, Imaeg Synthesis will be considered as the process of generating images from virtual 3D scenes using a computer. Computer graphics studios like Pixar and Dreamworks rely on propriety image synthesis pipelines to create photorealistic animations for their films. The final product that is released to the public is a direct result of the image synthesis process. Implementing a ray tracer is a daunting task but worth the effort for artists who are looking to make a career in professional computer-graphics lighting. Performing the steps required to create images from a ray tracer is invaluable to understanding how to optimize render time while still creating high quality images.

Writing a rendering program can be intimidating for an artist. Rendering programs consist of two fundamental parts: theory and implementation. The complex vector/matrix math theories needed to compute intersections within a 3D scene can be difficult to understand by themselves; coupled with proper programming language jargon can multiply the difficulty. In addition to remembering linear algebra mathematical processes, artists need to be concerned about code syntax and segmentation faults (generally an attempt to access memory that the CPU cannot physically address), which can be demoralizing to even the most experienced computer scientist. Another hurdle for artists is susceptibility to become distracted from the task at hand by focusing on the logistics of implementing code as opposed to gaining needed experience or experimenting with the mathematics of image synthesis.

Most students, when implementing a ray tracer, are siphoned into implementing their image synthesis programs in C++, perhaps because of tradition or familiarity

from previous class work. C++ can be a tricky language with many aspects of preparation needed before programming can begin. C++ programs can be executed extremely quickly, which is needed for *professional* image synthesis; the process of *learning* ray tracing theory, however, does not necessarily benefit from a high-speed processing program. This thesis has attempted to provide guidelines for introducing programming languages and techniques related to rendering so artists can spend less time on the implementation aspects of the process and more time on the theoretical experience of image synthesis. In addition to the C++ language, Processing, Python, and RenderMan© were evaluated for their potential as learning tools.

The qualitative results were generated by me, Christopher Potter. My background in scripting languages was informal prior to my admittance into the Visualization Department at Texas A&M. I hold a BFA from the Rochester Institute of Technology with a minor in Computer Science. My experience with computer languages, however, was not at a point where I felt comfortable writing simple scripts, nevermind an entire image synthesis program! Firsthand results recorded in this thesis are notes and "hiccups" encountered through each implementation step of the ray tracing process. My goal is that these results will demonstrate to student and teacher alike the complications that accompany each programming language and will help them overcome those obstacles in order to produce great art.

2. BACKGROUND: THE IMAGE SYNTHESIS PROCESS

3D rendering can be described as the process of creating 2D images from 3D geometric shapes with the use of photorealistic/nonphotorealistic effects using a computer. One method of 3D rendering is based upon the process of ray casting and ray tracing. For this thesis, ray casting and ray tracing is defined as the process of generating an image by tracing the path of virtual rays from an eye/camera source through pixels in an image plane and simulating the effects of the ray's interactions with virtual objects (Refer to Figure 2.1 for an illustration of the ray tracing process). Each ray-object interaction will be differentiated further by their illumination models, which is the shading algorithms used to interact with light rays. Four milestones have been established for the image synthesis process and out of those, three are associated directly with image synthesis theory: direct illumination, distributed ray tracing, and indirect illumination. The fourth milestone is preliminary preparations that are associated with each language used for this thesis.

2.1 Direct Illumination: Ray Casting

For this thesis, direct illumination encapsulates the ray casting process. Direct illumination will is referred to as illumination that emanates from a virtual light in a 3D scene that contributes to a 3D object's final color in addition to that object's material attributes. Direct illumination also encompasses an image's shadowing information, which is an obstruction in the light ray-object intersection.

Ray casting, a term first introduced by Scott Roth in 1982, is the process of casting rays into a 3D scene, finding the closest intersection from the eye/camera through boolean operations, and returning the color value of that object[19]. By emulating the different properties of light based on the cosine of the normal vector and the

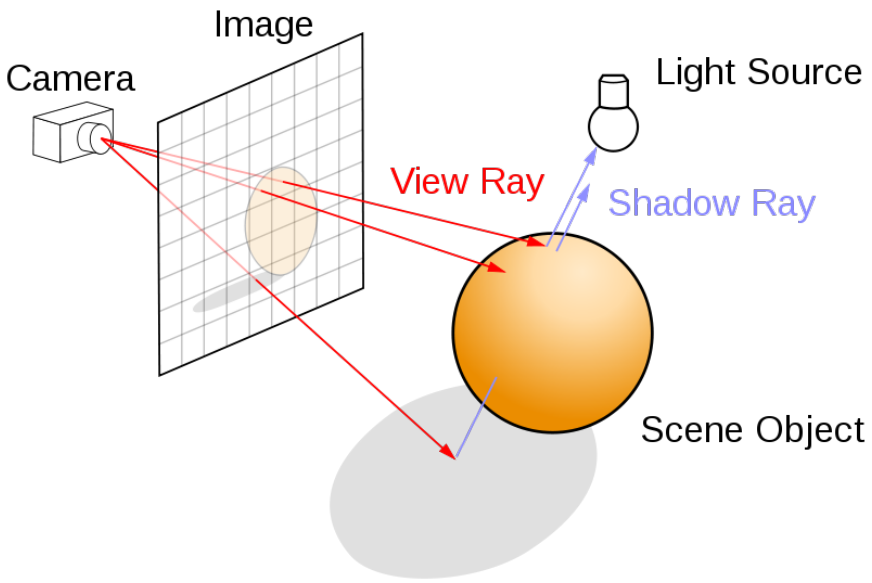

Figure 2.1: Ray tracing diagram [9]

normalized vector from the high point of the object to a light, semi-realistic results are produced. Other, more cartoon-ish, effects are created using similar methods, most notably Gooch and Gooch shading.

For this research, the basic Phong model of shading was used to generate specular highlights on 3D objects. This model, which has been altered slightly by Jim Blinn, will be discussed fully in following sections. The Phong model equation is for a ray intersecting the surface of an object at point P as follows:

$$color_{output} = k_d O_d I_a + k_d O_d I_d[\hat{L_m} \cdot \hat{N}] + k_s O_s[\hat{R_m} \cdot \hat{V}]^n I_d[11] \tag{2.1}$$

where k_d and k_s are the diffuse and specular reflection coefficients, I_a is the ambient light color, I_d is the diffuse light color, O_d is the diffuse color of the object, L_m is the direction vector from point P towards the light, N is the surface normal for the surface at P, O_s is the specular color of the object, R_m is the direction that a

perfectly reflected ray of light would take from P, and n is the specular exponent, or shininess. Each part of this equation will be discussed. The second part of the equation, $k_d O_d I_d[\hat{L_m} \cdot \hat{N}]$, can be easily defined with Lambert shading.

2.1.1 Lambert Shading

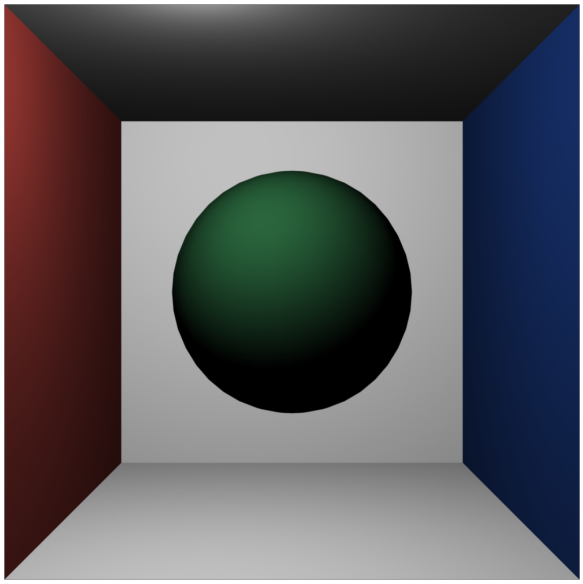

Figure 2.2: An example of lambert shading created in Autodesk Maya 2010©

The Lambertian model for reflectance is the diffuse component of a surface's material, or the diffuse reflection of a material's properties. Diffuse objects are generally thought of as anything that does not reflect light off of its surface, such as unfinished wood. This effect can be seen in Figure 2.2. This model follows Lambert's cosine law for optics, which states that the radian intensity or luminous intensity observed from an ideal diffusely reflecting surface or ideal diffuse radiator is directly proportional to the cosine of the angle θ between the observer's line of sight and the surface normal. Applying this law to computer graphics we the following

equation:

$$color_{output} = \hat{L_m} \cdot \hat{N} C I_L \tag{2.2}$$

where $\hat{L_m}$ is the normalized direction of the light-direction vector, $\hat{N}$ is the normalized normal vector, C is the color, and I_L is the intensity of the incoming light. Since $L \cdot N = |N||L| \cos\theta$, if we make the lengths of $|N||L| = 1$ by normalizing them, equation 2.2 satisfies the properties for Lambertian shading by making $L \cdot N = \cos(\theta)$. As we can see, Equation 2.2 is actually the term represented as $k_d O_d I_d[\hat{L_m} \cdot \hat{N}]$ in Equation 2.1, where $C = O_d$ and $I_L = I_d$.

2.1.2 Gouraud Shading

One of the first shading paradigms introduced to the computer graphics community was a vertex interpolation shading created by Henrik Gouraud at the University of Utah[8]. Gouraud's algorithm successfully captured the Lambertian Shading effect described in Section 2.1.1. Gouraud's approach saved intensity values at each vertex as a weighted average of the normals of the surrounding polygons in the object's mesh. When a hit on an object was achieved, a weighted sum of the point's color was returned depending on the intensity values at each closest vertex. This effect is demonstrated in Figure 2.3.

While this method works well, it is also dependent on the density of the object's mesh since each vertex intensity is basically an average of the surrounding normals. This technique also benefits from the relatively simple calculations needed per hit point because, generally speaking, a polygon is made up of a 3 or 4 vertexes maximum. Instead of calculating Equation 2.1 at each pixel of intersection within the scene, interpolation between precalculated vertex intensities is needed. Unfortunately, since the smooth perception of shading is based off of interpolation, which

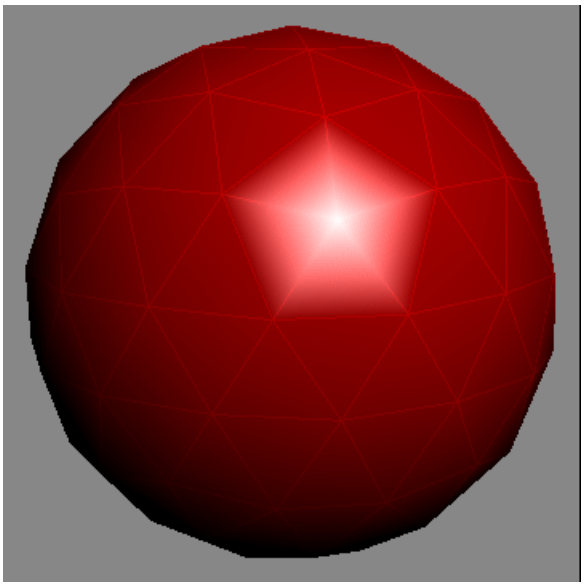

Figure 2.3: An example of the deficit of gouraud shading being apparent in the highlight in a lowpoly model. [12]

in effect is characterized by the density of object meshes, high localized specular highlights will not be rendered correctly. Also, if a highlight lies in the middle of a polygon, but does not spread to the polygon's vertex, it will not be apparent in the render. If the highlight appears directly on a vertex, while it will be rendered correctly for that vertex, it will be rendered incorrectly on neighboring polygons. An example of this deficit for Gouraud's shading can be see in Figure 2.3. For this reason, Equation 2.1 was introduced.

2.1.3 Phong Shading

Another aspect of an object's illumination model is the specular component of an object. Specularity is the visual appearance of specular reflections. This represents an object's "shininess". One strategy to display this specular highlight was introduced by Bui Tuong Phong [18]. Whereas Phong's predecessors interpolated across surface patches [8], as stated in Section 2.1.2, Phong interpolated surface normals

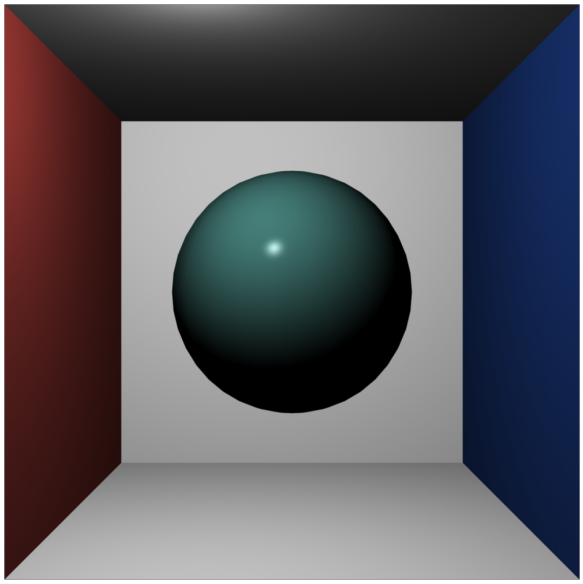

Figure 2.4: Phong shading created in Autodesk Maya©2010

and evaluated a lighting model at each pixel. He also added a specular component to the lighting model to produce highlights.[13]

Phong's specular component from his paper *Illumination for Computer-Generated Images* introduces the equation at ray-object intersection point P:

$$color = C_p[\cos i + d] + W(i)[\cos s]^n \tag{2.3}$$

where C_p is the reflection coefficient of the object at point P, i is the incident angle, d is the environmental diffuse reflection coefficient, $W(i)$ is a function which gives the ratio of the specular reflected light and the incident light as a function of the incident angle i, and s is the angle between the direction of the reflected light and the light of sight.

The important aspect of Equation 2.3 is the term $[\cos(s)]^n$. Since the cosine of the angle between the direction of the reflected light, or R_m in Equation 2.1, and

the line of sight, V, is equal to $R_m \cdot V$ as long as each vector is of unit length, we only need to know what R_m is.

To calculate R_m we have this equation from Phong as well[13] at ray-object intersection point P:

$$R_m = 2[L_m \cdot N]N - L_m \tag{2.4}$$

where L_m is the direction vector from P towards the light and N is the surface normal at P. The effect of Phong shading can be seen in Figure 2.4.

2.1.4 Blinn Shading

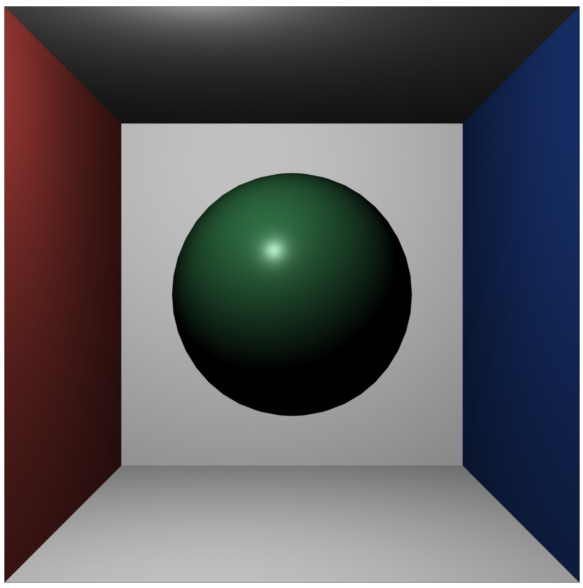

Figure 2.5: Blinn shading created in Autodesk Maya 2010©

In Jim Blinn's paper *Models of Light Reflection for Computer Synthesized Pictures* he introduces the equation [1]:

$$color = p_a + max(0, N \cdot L)p_d + (N \cdot H)^n p_s \tag{2.5}$$

which is most similar to Equation 2.1 because of its ambient term p_a, diffuse term p_d and specular term p_s. Most importantly for Blinn's equation is the inclusion of H. Instead of using Phong's R_m from Equation 2.4, Blinn introduces a new half-vector called H. Blinn describes,

> "If the surface was a perfect mirror, light would only reach the eye if the surface normal, N, pointed halfway between the source direction, L, and the eye direction, E. We will name this direction of maximum hilights H..."

Relating this to the variables in this paper:

$$H = \frac{\hat{L} + \hat{V}}{len(L + V)} \tag{2.6}$$

Equation 2.6 then replaces $R \cdot V$ in Equation 2.1. Blinn's model produces more accurate models of empirically determined bidirectional reflectance distribution functions for many different types of surfaces[17]. While this equation produces better results, it also introduces a square root math function when determining $len(L+V)$. Since a square root calculation takes more time than a dot product calculation, Blinn's model has been considered slower.(With computer processing achievements today, however, the speed differences are incomparable.) Although it will not be used in this thesis, I hvae included it to help show the origins of Equation 2.1. The effect of Blinn shading can be seen in Figure 2.5.

2.1.5 Gooch and Gooch Shading

Ever since the introduction of Phong's model for photo-realistic rendering of geometric objects, there has been a trend towards non-photorealistic rendering(NPR). Most notable in this field is the work of Amy and Bruce Gooch, et al. Gooch and

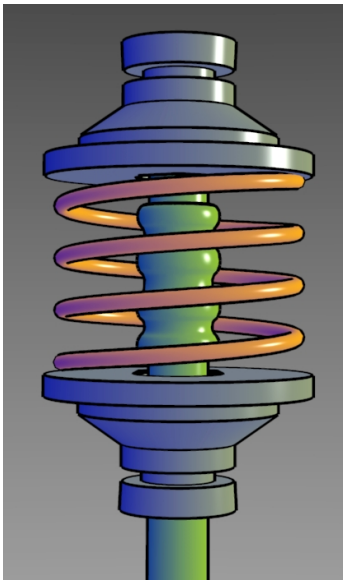

Figure 2.6: Gooch shading example [6]

Gooch argue that "Phong shaded 3D imagery does not provide geometric information of the same richness as human-drawn technical illustration[6]." NPR techniques are referred to as computer graphics algorithms that imitate traditional techniques such as painting or pen-and-ink[6]. Gooch, et al. introduced a generalization of the classic computer graphics equation from Equation 2.2 by experimenting with the value of $L_m \cdot N$. Their equation is as follows:

$$color = \left(\frac{1 + (\hat{L_m} \cdot \hat{N})}{2} \right) k_{cool} + \left(1 - \frac{1 + (\hat{L_m} \cdot \hat{N})}{2} \right) k_{warm} \tag{2.7}$$

where k_{cool} and k_{warm} are two color values to interpolate between. In Equation 2.7, they use the large fractions to remap the values of $\hat{L_m} \cdot \hat{N}$ between zero and one, causing the colors to gracefully blend from k_{cool} to k_{warm}. As can be seen in Figure 2.6, the Gooch shading algorithm also includes outlines around each shape. Gooch referred to creating the outline in the paper *Real-Time Nonphotorealistic*

Rendering[14], but for this thesis the outline is determined from when the values of $\hat{L_m} \cdot \hat{N}$, after being remapped between 0 and 1, are between a certain threshold and zero. While this may not the best way to generate outlines around 3D shapes, it accomplishes semi-reliable results and was therefore utilized in this thesis.

2.1.6 Texture Mapping and UV Coordinates

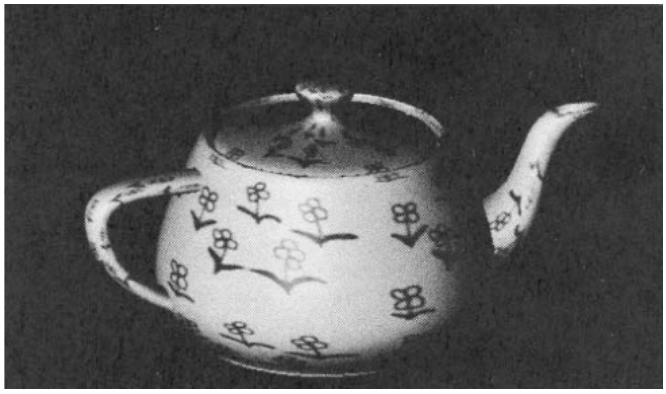

Figure 2.7: An example of texture mapping to the surface of a teapot. [3]

Up to this point, we've focused on explaining basic color appropriation with solid colors represented as RGB; in computer graphics,however, it is also possible to map photographs to the surface of 3D objects. This idea was first introduced by Ed Catmull in 1974 when he demonstrated a method of representing 3D curved patches as opposed to conventional polygons. This method "maps" photographs to the surfaces of these patches[5]. Jim Blinn expanded this to include environmental mapping based off of reflections from the surface of objects[3], as can be seen in Figure 2.7.

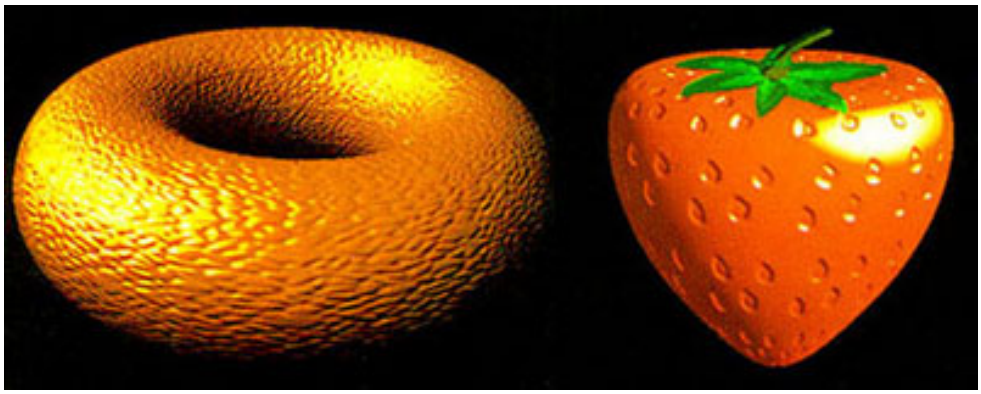

Figure 2.8: An example of bump mapping introduced by Jim Blinn. [2]

2.1.6.1 Bump/Normal Mapping

As another expansion of Catmull's texture mapping, Blinn also introduced a technique that could simulate a rough or textured surface on a 3D shape, called bump mapping[3]. Blinn discovered that by altering the normal at each point of intersection on the surface of a 3D shape, simulations of wrinkled surfaces can be generated as seen in Figure 2.8. The normal at each point can be mapped to an image texture or altered via a noise generating equation such as Perlin noise. Blinn's wrinkled surface simulation can be seen in Figure 2.8.

2.2 Ray Tracing and Distributed Ray Tracing

To improve upon the Phong specular shading model, a new recursive technique was introduced by Turner Whitted at Bell Laboratories. Turner hypothesized that a tree model should be used to calculate accurate, realistic representations of the physical world's reflective process. He based his algorithm on an reflection ray, R, that represents the reflection of a ray on a perfectly smooth surface, and P, which represents the transmitted ray through a perfectly smooth surface:

$$\hat{V}' = \frac{\hat{V}}{\hat{V} \cdot \hat{N}} \tag{2.8}$$

$$\hat{R} = \hat{V}' + 2\hat{N} \tag{2.9}$$

$$\hat{P} = k_f(\hat{N} + \hat{V}') - \hat{N} \tag{2.10}$$

where

$$k_f = (k_n^2|\hat{V}'|^2 - |\hat{V}' + \hat{N}'|^2)^{\frac{1}{2}} \tag{2.11}$$

and k_n is the index of refraction for the surface. The equations assume that $\hat{V} \cdot \hat{N}$ is less than zero so the sign of N must also be adjusted to point to the side of the surface the intersecting angle is incident from. When tracing a ray from the eye point, the intersection at each reflective surface determines the next surface hit, forming a recursive tree formation. Upon achieving this tree, the following equation calculates the surface color:

$$color_{output} = I_a + k_d \sum j = 1^{j=ls}(\hat{N} \cdot \hat{L_j}) + k_s S + k_t T \tag{2.12}$$

where

S = the intensity of light incident from the $\hat{R}$ direction

k_t = the transmission coefficient

T = the intensity of light from the $\hat{P}$ direction

By keeping k_s and k_t constant Turner achieved his results, but for ideal circumstances the values should be mapped to a Fresnel algorithm that relates them to realistic models of reflection and refraction.

To enhance the raytracing process, Cook, Loren and Carpenter introduced the

term “distributed ray tracing”, described as:

> “...The key is that no extra rays are needed beyond those used for over-sampling in space. For example, rather than taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations are traced at different instants of time.
>
> - Sampling the reflected ray according to the specular distribution function produces gloss (blurred reflection)
> - Sampling the transmitted ray produces translucency (blurred transparency).
> - Sampling the solid angle of the light sources produces penumbras.
> - Sampling the camera lens area produces depth of field.
> - Sampling in time produces motion blur.”

The introduction of randomization, or jittering , achieves the distributed ray tracing effect. By jittering the R value of T in Equation 2.12 a glossy effect is produced. The same can be said of the P value from the T variable in Equation 2.12. Distributed ray tracing is important to a 3D image’s realism because it is not possible to achieve the computed perfect reflection/refraction model introduced by Whitted in the natural world.

2.3 Indirect Illumination/ Radiosity Effects

For this thesis, Indirect illumination accounts for any shading or color values not calculated directly from the virtual light in a scene. These terms are known as ambient occlusion, global illumination and caustic effects. In the natural world, diffuse reflection is the reflection of light from a surface such that an incident ray is

reflected at many angles rather than at just one angle, which is the case of specular reflection. Typically an object will base its final color off of not only the light shining at it, but also from the colors of the objects in closest proximity to it. This effect is seen in Figure 2.9 in the picture of a racquetball. The surface of both the racquetball and the piece of paper are as close to Lambertian surfaces as can be found in the natural world. In the figure the diffuse reflection is most prominent in the racquetball's shadow. It may seem like a trick on the eyes, but the shadow has a purple tint to it because of the diffusely reflected light rays coming from the racquetball. In theory, if an object is illuminated in a room with bright red walls the color of the object will have a red tint because of the diffuse reflection of the red from the walls contributing to the overall color of its surface. It is near impossible to determine all the different contributors to an objects final color in the natural world because of the infinite amount of light rays absorbed and reflected by an object's surface.

To simulate this effect in a 3D environment, new terms were introduced by Torrance, Greenberg et.al [7], who determined a model of light interaction between diffuse surfaces based off of methods used in thermal engineering. To simplify their method, this thesis used a method inspired by their work. The process to calculate ambient occulsion, global illumination, and caustic effects are fundamentally all the same. At each point in a scene, a specific number of sample rays will be cast at each intersection point. The average of the resulting color information from each sampling ray set determines that pixel's final color. Ambient occlusion is the simplest. Ambient occlusion informs the proximity of objects with other objects. Ambient Occlusions produces a black and white image that will be darker in pixels where images are closer together and lighter in places where images are farther apart. To calculate this set, one simply needs to determine if another object is hit by a ray in

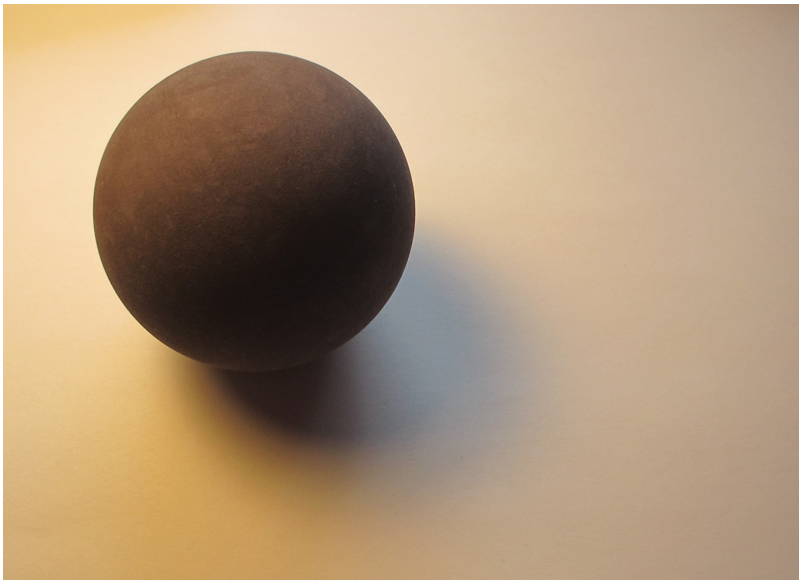

Figure 2.9: Real world indirect illumination/radiosity effect with a racquetball on a piece of paper.

the set or not. For instance, if 255 rays are cast from an object, and 200 of them hit another object in a distance greater than 0, then the value for that pixel will be 200/255 or .7843. An ambient occlusion example can be seen in Figure 2.10.

To calculate global illumination, at each point in the sample set of data, instead of calculating the distance from the point, the direct illumination shading value is added. Rather than the average number of intersections, the average color is determined from 255 sample rays for a global illumination calculation. This is seen in Figure 2.11. This process is where the color bleeding from nearby objects can best be seen.

The final effect is a caustic, which is the ratio of specular color at each point on a surface. If a surface receives caustics, it calculates the reflective rays of nearby objects and returns the reflected color value average for each sample ray set. This effect is seen in Figure 2.12.

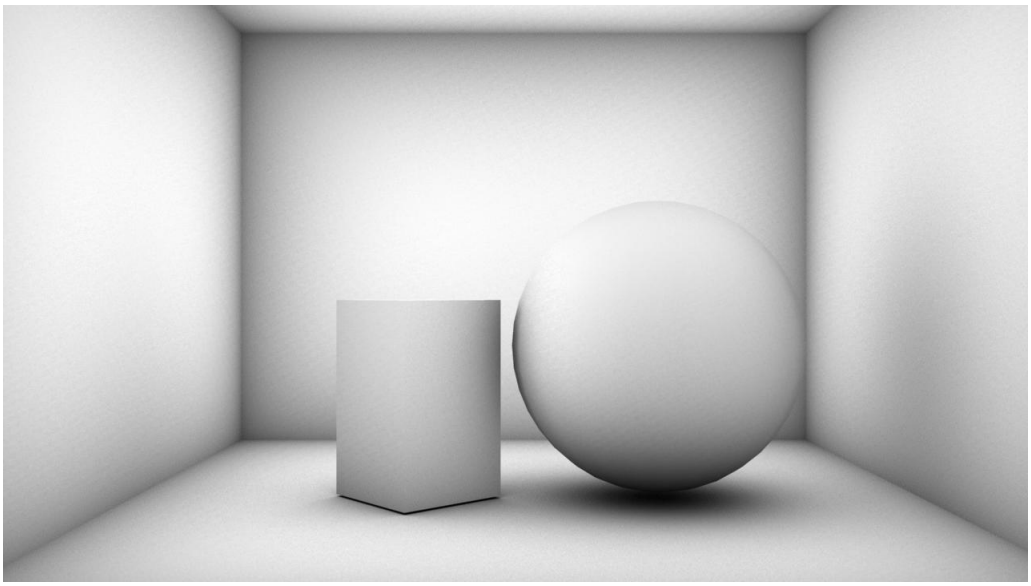

Figure 2.10: Example of ambient occlusion

Indirect illumination combined with distributed ray tracing has made a difference in the realistic perception of generated photorealistic images, and an effort is being made to incorporate these effects into realtime settings like video games in order to provide a more immersive experience. Large sample ray sets are needed for more realistic looking images however, and the computing power required to produce this effect in realtime is sizable and expensive.

2.4 Languages

The four languages I evaluated were determined for specific reasons. They had to have the ability to follow the object-oriented paradigm and possess the potential to perform graphical methods like imaged reading and writing. A brief history behind each language will be discussed to give insight into the basis for each language's creation, which will also provide insight as to their feasibility in creating an image synthesis program.

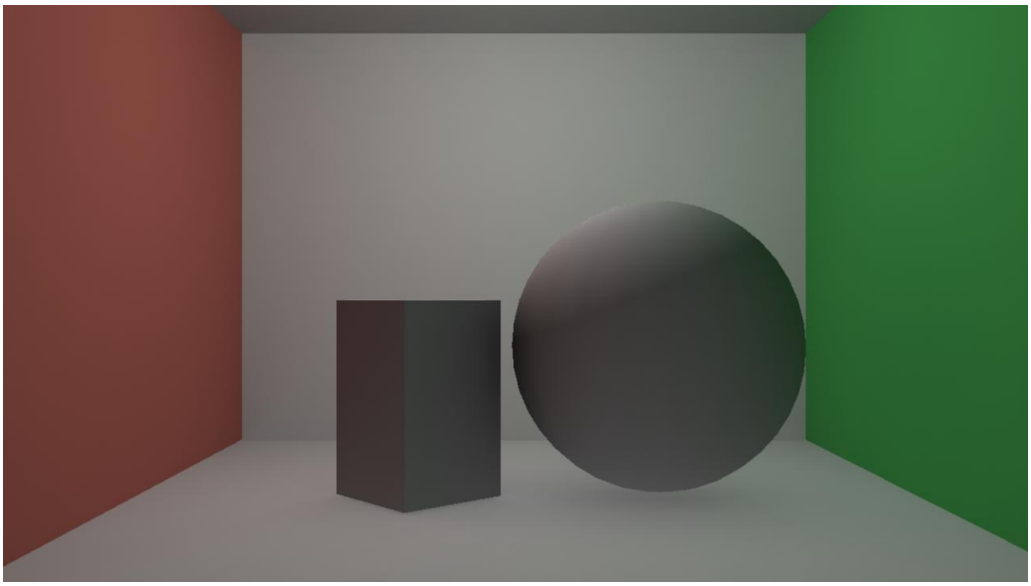

Figure 2.11: Example of global illumination

2.4.1 C++

C++ is a general-purpose object-oriented coding language. Its genesis lies in research done with the programming *C with Classes* in 1979, although its commercial release was not until October 1985[20]. C++ is considered an intermediate-level language because of its capabilities with both high and low level computer functionality. Creator Bjarne Stroustrup claims to have drawn inspiration from not only C, but also from Simula, Algol68, BCPL, Ada, CLU and ML. Since its creation, C++ has gone on to influence the creation of numerous coding languages, such as C#[16].

According to Stroustrup, C++ was originally designed to combine Simula's facilities for program organization with C's efficiency and flexibility for systems programming[20]. C++ is an extension of the C programming language. In fact, it stemmed from a fundamental design called *C with Classes*[20]. When Stroustrup designed the C++ language he had a variety of guidelines he deemed as "suitable" for computer lan-

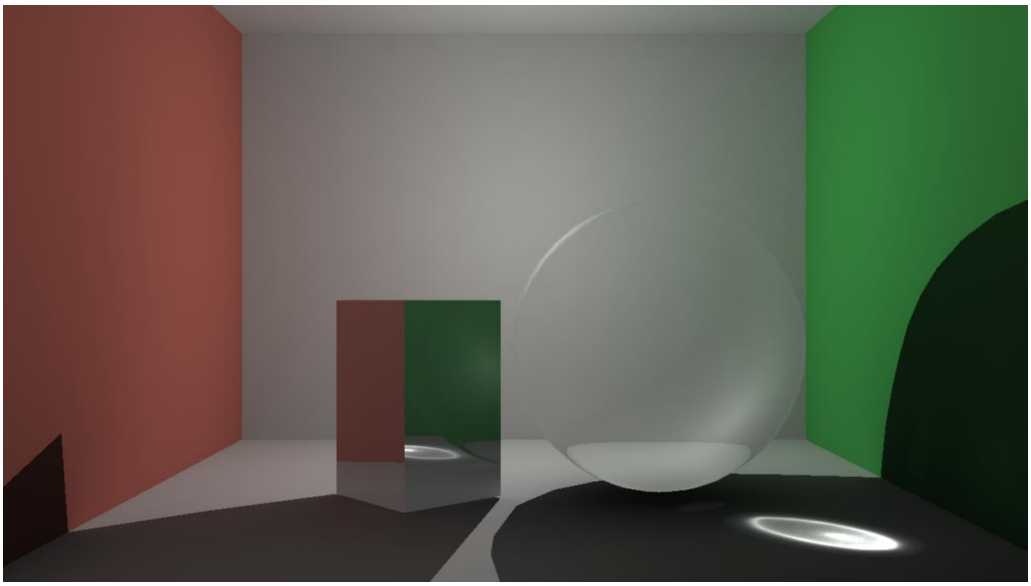

Figure 2.12: Example of caustic effect

guages. He writes:

[1] " A good tool would have Simula's support for program organization- that is, classes, some form of class hierarchies, some form of support for concurrency, and strong(that is, static) checking of a type system based on classes. This I saw as support for the process of inventing programs, as support for design rather than just support for implementation.

[2] A good tool would produce programs that ran as fast as BCPL programs and share BCPL's ability to easily combine separately compiled units into a program. A simple linkage convention is essential for combining units written in languages such as C, Algol168, Fortran, BCPL, assembler, etc., into a single program and thus not get caught by inherent limitation in a single language

[3] A good tool should also allow for highly portable implementations. My experience was the "good" implementation I needed would typically not be available until "next" year and only on a machine I couldn't afford....[20]"

To further emphasize C++'s benefits, Stroustrup explained why he chose C over other languages of the time to build upon:

"C is clearly not the cleanest language ever designed nor the easiest to use so why do so many people use it?

[1] C is *flexible*: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.

[2] C is *efficient*:The semantics of C are "low level"; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.

[3] C is *available*: Given a computer, whether the tiniest micro or the largest super-computer, the chance is that there is an acceptable quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also library and support tools available, so that a programmer rarely needs to design a new system from scratch.

[4] C is *portable*: A C program is not automatically portable from one

> machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependencies is typically technically and economically feasible.
>
> Compared with these "first order" advantages, the "second order" drawbacks like the curious C declarator syntax and the lack of safety of some language constructs become less important. Designing "a better C" implies compensating for the major problems involved in writing, debugging, and maintaining C programs without compromising the advantages of C. C++ preserves all these advantages and compatibility with C at the cost of abandoning claims to perfection and of some compiler and language complexity. However, designing a language "from scratch" does not ensure perfection, and the C++ compilers compare favorably in runtime, have better error detection and reporting, and equal the C compilers in code quality. [20]"

Stroustrup sought to create a universal, object-oriented language that was accessible and relatively low maintenance to begin programming with since most computers were, and still are, compatible with C.

Object-oriented programming provides clean and modular code that is easier to maintain and debug, because it separates functions and utilities into class objects, which can be reused. This is C++'s main benefit to the image synthesis process, in addition to its efficiency in utilizing hardware resources from a computer to provide faster results.

2.4.2 Processing

Processing is an open-source programming language and integrated development environment, or IDE, developed by Casey Reas and Benjamin Fry in 2001. According to the Processing website (processing.org):

> "Processing is a programming language, development environment, and online community. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. Initially created to serve as a software sketchbook and to teach computer programming within a visual context, Processing evolved into a development tool for professionals. Today there are tens of thousands of students, artists, designers, researchers and hobbyists who use Processing for learning, prototyping and production."

Andrew Glassner, a pioneer in Ray Tracing, also wrote a book on Processing called *Processing for Visual Artists*. Glassner helps to emphasize the importance and use for Processing in the visual world:

> "Processing is for artists, designers, visualization creators, hobbyists or anyone else looking to create images, animation, and interactive pieces for art, education, science or business....Processing offers you a 21st-century medium for expressing new kinds of ideas and engaging audiences in new ways..."

The usefulness and applicability of Processing in the ray tracing process can be found in its mission statement. It was initially designed to teach computer programming within a visual context. Since that is also the aim and goal of this thesis, Processing was a reasonable option to investigate. Processing uses Java syntax and

packages the java compilation process into its IDE, so a functional overview of Java is also needed to fully determine the usefullness of Processing.

2.4.2.1 Java

The book *Core Java 2: Volume 1- Fundamentals*[10] identifies eleven buzzwords used by the authors of Java: simple, object oriented, distributed, robust, secure, architecture neutral, portable, interpreted, high performance multithreaded and dynamic.[10] I will not review all of these terms. The most prevalent terms are simple and architecture neutral. The rest of the terms touch low level mechanics of the language that were not considered when deciding to implement a ray tracer.

Core Java 2: Volume 1- Fundamentals summarizes the buzzword simple as follows after comparing it with C++:

> "The syntax for Java is, indeed, a cleaned-up version of the syntax for C++. There is no need for header files, pointer arithmetic (or even a pointer syntax), structures, unions, operator overloading, virtual base classes, and so on. (See the C++ notes interspersed throughout the text for more on the differences between Java and C++.) The designers did not, however, attempt to fix all of the clumsy features of C++. For example, the syntax of the switch statement is unchanged in Java. If you know C++, you will find the transition to the Java syntax easy."

One of the intents of Java was to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice.[10] It seemed that Java was intended to be a simpler of version of C++, which is one of the reasons it was used for this thesis. *Core Java 2: Volume 1- Fundamentals* summarizes architecture neutral as well:

> "The designers of Java did an excellent job developing a bytecode instruction set that works well on today's most common computer architectures. And the codes have been designed to translate easily into actual machine instructions."

Java was also intended to be simple to run in any operating system, as long as the Java run time system existed.[10] This was also one of the reasons Java was chosen.

While a majority of the material I used for the Processing portion of the project was directly related to Java, it is important to mention a comment from the creator of Processing, Ben Fry on his blog found at http://benfry.com/writing/archives/169:

> "However, we dont print Java on every page of Processing.org for a very specific reason: knowing its Java behind the scenes doesnt actually help our audience. In fact, it usually causes more trouble than not because people expect it to behave exactly like Java. Weve had a number of people who copy and pasted code from the Java Tutorial into the PDE, and are confused when it doesnt work...
>
> ...But for as much trouble as the preprocessor and language component of Processing is for us to develop (or as irrelevant it might seem to programmers who already code in Java), were still not willing to give that updamned if were gonna make students learn how to write a method declaration and public class Blah extends PApplet before they can get something to show up on the screen."

Even though Fry suggests that it is not important to understand the Java language to begin using the Processing IDE, I found it very helpful to understand how the Java language handles variables when writing the ray tracers since the written programs are more complex than simple Processing sketch. Another interesting

fact to point out can be found in another quote from on the wiki page for Processing(https://github.com/processing/processing/wiki/FAQ):

> "We didn't set out to make the ultimate language for visual programming, we set out to make something that was:
>
> - A sketchbook for our own work, simplifying the majority of tasks that we undertake,
> - A teaching environment for that kind of process, and
> - A point of transition to more complicated or difficult languages like full-blown Java or C++ (a gateway drug to more geekier and more difficult things)
> - At the intersection of these points is a tradeoff between speed and simplicity of use. i.e. if we didn't care about speed, Python, Ruby or many other scripting languages would make far more sense (especially for the simplicity and education aspect of it). If we didn't care about transition to more advanced languages, we'd get rid of the C-style (well, Algol, really) syntax. etc etc.
>
> Java makes a nice starting point for a sketching language because it's far more forgiving than C++ and also allows users to export sketches for distribution across many different platforms. When we got started in 2001, most people were using it to build applets that ran on the web, which was important to the early growth of the project"

The Processing language is not based off of the Java language in a one to one ratio. The biggest influence over the creation of the Processing language was the

simplify the Java implementation process to make it quicker to develop with, and act as an introduction or transition into the C++ development world.

2.4.3 Python

Python, named after *Monty Python's Flying Circus*, was first founded by Guido van Rossum. He began his work on Python at the the National Research Institute for Mathematics and Computer Science in the Netherlands in 1989. Python is a high-level and interpreted programming language. While it can be argued that all computer languages are interpreted, Python is considered an interpreted language because unlike C or C++, Python does not require a compiler to operate. While C and C++ compiled code needs to be compiled into machine-language before it is relayed to the computer, Python's instructions are interpreted directly from its written code. van Rossum is quoted as saying he was unhappy with the productivity of creating a script or utility in C, which influenced his interest in establishing Python[21].

One of the main focuses of Python is readable syntax. According to Jim Mcconnell's book *Code Complete*, one line of Python is equivalent to six lines of C code[15]. According to van Rossum, Python's creation was heavily influenced by the coding language ABC. ABC's design was intended to be a programming language that could be taught to intelligent computer users who were not computer programmers or software developers. The main deficit with ABC's design was its inability to bridge the gap in GUI creation and an inability to directly access the file system and operating system in a computer. Python's syntax eliminates the need for traditional curly braces ({}) and instead uses a tabular system that denotes code blocks.

2.4.4 RenderMan

RenderMan© is a software and application programming interface, or API, that many companies in the computer industry use to render large projects for entertainment or video game use. RenderMan© specializes in network distribution of renderings throughout a "renderfarm" that has the ability to render ray traced images faster than a single computer. RenderMan© is referred to as the rendering engine that produces the image. RenderMan© is dependent on two files: RenderMan© Interface Bytestream (RIB) and RenderMan© shading file. RIB acts as a descriptor for how the engine should work, incorporating all aspects of ray casting, ray tracing, direct illumination, distributed ray casting, and direct illumination. A shading file is written in the RenderMan© Shading Language (RSL), and acts as a description file for how objects in a scene interact with light. By utilizing the RIB and RSL files, each ray tracing milestone was accomplished within the scope of this thesis.

It is important to note that RenderMan© has already implemented all of the milestones needed for this thesis. Also, RenderMan© is based off of a camera projection matrix algorithm that does not cast rays the same way that this thesis does ray casting. RenderMan© also incorporates a variety of highly advanced rendering techniques that are too far outside the scope of this modest master's thesis to discuss. The thesis will discuss the feasibility of utilizing the pre-constructed implementations of the milestones without the aid of production software such as Autodesk Maya© or Side Effects Houdini© . This will present the same types of implementation issues that C++, Processing, and Python would in regards to the theory of ray casting.

3. METHODOLOGY

The methods I used were straightforward. For each programming language notes were kept that outlined the difficulties and roadblocks caused specific to the language. The same programming theories and fundamentals were utilized in order to provide continuity between each programming process. Due to its helpfulness in managing and organizing large coding projects, the object-oriented approach to programming was taken. This means that classes with specific variables and data-structures were defined so that instances of objects could be constructed at runtime. Wherever possible, the image synthesis process was sub-categorized in order to better organize the information needed for the process. This led to shorter development time and also contributed to quicker troubleshooting after the code was written.

The program written for each language used a similar data structure and was modelled after the same overall design. For each language a similar data structure was also strived for, shown in Figure 4.6.

A dynamic data structure that resizes as objects are added or removed along with an agnostic data type was desired for this project. As we discussed next section, this approach provided a solution that mirrored the functionality of a ray caster.

In addition to having a program architecture and data structure, the milestones I established that segment image synthesis theory within four categories also provided a structured approach to analyzing and reporting the conclusions used for each language. Since each program was modeled after this conceptualization, measurements of success and difficulty were more clearly defined. More details on the milestones are discussed in the following section.

By determining the level of difficulty of implementing the program's data struc-

tures and class design, not to mention using that design to further implement the image synthesis milestones of Preliminary Preparations, Direct Illumination, Ray Tracing/Distributed Ray Tracing, and Indirect Illumination, a detailed report has been compiled that informs the results collected and reported in this thesis, all found in the next section.

4. IMPLEMENTATION RESULTS

4.1 Image Synthesis Program Structure and Implementation

The basis for every ray tracing program can be broken down into a simple set of processes. First, a program must be able to read and write images. Once the ability to read and write images has been realized, the process for casting rays can begin. This process is demonstrated by the following pseudocode, or code outline:

Algorithm 1: Ray Casting Pseudocode

```
foreach pixel in image do
    foreach object in scene do
        if object intersects ray then
            intersectionDistance = distance of viewpoint from intersection of
            object and ray;
            if intersection > previousIntersection then
                previousIntersection = intersection;
            end
        end
    end
    if intersection exists then
        foreach Light in scene do
            determine materialColor from light;
        end
        pixel = materialColor;
    end
    else
        pixel = black or alpha(no color)
    end
end
write image;
```

Each pixel in an image will correspond to a position in 3D space, as demonstrated

in Figure 2.1, found on page 4. In this figure, each box represents an image pixel. To determine each pixel's final color, one must first determine if there is an intersection with any of the 3D objects in a scene and if so, which object is closest. Then, that object's illumination algorithm must be run, factoring in the light source(s) in the scene. Each ray tracer written followed these steps, as detailed in Algorithm 1, but each ray tracer also adheres to specific structure and code organization. In order to better understand the ray tracing structure, I needed to establish and learn some important Computer Science terms as part of the preliminary preparations for the project.

4.2 Milestone 1: Preliminary Preparations

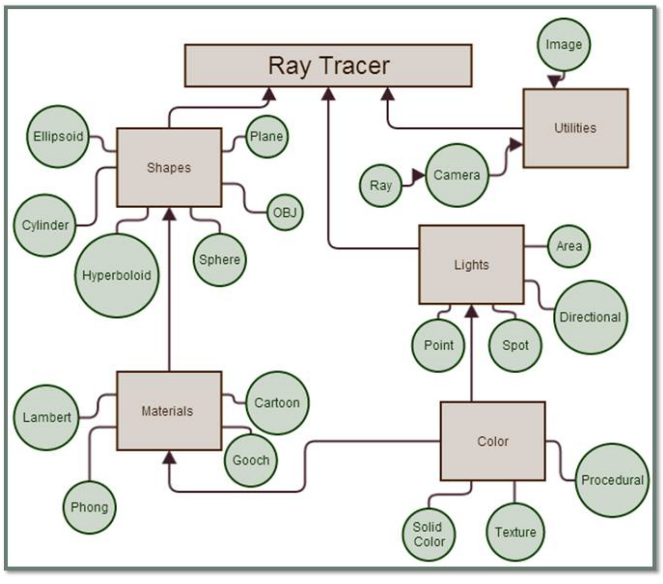

Figure 4.1: Diagram of the parenting/dependency structure of the ray casting program

Preliminary preparations refers to anything that needed to be accomplished be-

fore being able to start coding the ray tracing theory coding. In this section, simple matters such as program language installation and accessibility will be discussed. Also discussed is the majority of the "Utilities" section of Figure 4.1.

The foundations of a ray tracing program are built around two fundamental procedures: image writing and vector math. Ray tracing can also be accomplished through matrix operations, but for the purposes of this thesis, vector math will be sufficient. Vector math is the basis for all ray tracing theory. As such, a class or data structure that can accomplish all aspects of vector math including addition, subtraction, dot product, and cross product is crucial to accomplishing the image synthesis program. In addition to this, in order to see the final product after all the calculations have been run, you must be able to write an image. Arguably the easiest image file to write out is a Portable PixMap (PPM). Other image formats include JPEG, BITMAP, and PNG. Each of these image formats requires a more complicated compression algorithm than PPM. Each programming language may also require additional files in order to compile or execute the final product. Each of these languages will be assessed from the viewpoint of a student in the Visualization Department of Texas A&M, because this is directly relevant to the my experience.

4.2.1 C++

The C++ language, as described in Section 2.4.1, is an object-oriented language that is considered both a high and low level language because of the level of control it provides over computer functionalities. Development and installation with C++ was accomplished via the provided computers at Texas A&M's Department of Visualization.

When developing in a Linux/Unix environment, C++ can be executed directly from the terminal of the operating system. This means no external IDE is needed

to compile your code. This development environment was more appealing to me because IDE environments have their own learning curve, which would have to be realized in addition to learning the C++ language semantics. If a Department of Visualization computer was not available, installation of the C++ development tools would present its own challenges.

To work on a Windows computer, you must first determine which compiler to work from, download that compiler, and then work from that specific compiler's commandline. Common practice would be to download an IDE such as Visual Studio or Eclipse. Since computers are not typically sold with a Linux operating system installed on them, developing in Linux outside of an academic setting would require installing a new operating system, which is not a task for beginners. After learning how to install Linux, you must then learn the proper way to download specific compiler packages. Since this can be accomplished by simply typing a command into the computer's terminal, it can be significantly easier to learn how to do than installing a compiler on Windows. A Macintosh computer was not tested, but since their operating systems are a variation of the Unix platform much like Linux, and since they are more commonly sold in stores, they are more accessible to students who are not computer scientists. Starting to develop in C++ on any computer is a complicated task unless a ready-made environment has been provided for you, like at Texas A&M.

What makes C++ unique from other languages explored in this thesis is the concept of a makefile, and the make command. In order to communicate with the compiler installed on the Unix platform, you need to create a file that will define which source code files are needed to make object files and which compilers and linkers are included in the make command. A properly configured IDE it will automatically generate a makefile and perform the make command for you. This further solidified

my decision to work without a IDE and use a syntax highlighting text-editor, because nothing was generated automatically. That way, only what was typed and developed by myself was included in the project. This allowed for easier debugging of the makefile and make command rather than debugging the IDE's internal processes.

The use of header files is another difference between C++ and the other languages used in this thesis.. When creating a class in C++, it is common practice to outline the class structure in one of these "header" files. The header file serves as the skeleton of the class structure, while the meat of the class's implementation is fleshed out in a ".cpp" file. This is a technique inherited from C coding which was used for older computers that could not support large quantities of memory at the same time. Computers have improved since and this antiquated practice now further segregates the interface of a class from the implementation of the class rather than save memory space during execution time. For this thesis only header files were used since speed optimization was not a priority.

In addition to a makefile, another obstacle introduced by C++ was the vector math library. C++ does not have a standard vector library: the operation needs to be hand-coded in order to be available for use. A developer has to either implement their own vector math library or borrow a pre-existing library. This requires additional research or work to write the necessary code. If a developer is going to borrow code from the Internet, an assessment needs to be made regarding the original writer's accuracy. Subsequently, knowledge must be gained on how to use the classes and structures defined in the borrowed library. The vector class used for this thesis was a library circulating amongst the Texas A&M students that was written by a former faculty member at A&M: the know source is reputable and the code is trusted to be uselful and correct.

The same obstacle applies to the image reading and writing functionality of an

image synthesis program. There is no standard C++ image library. Once again, in order to read and write images, a developer needs to either write their own code or borrow an existing third-party library. For this thesis, a class was written to process .ppm images. I decided that the effort required to create a custom class to process .ppm images would be less than the effort required to untangle the semantics of linking in a third-party image library. The .ppm format was chosen because it was the most accessible and simplest format to implement. The created images needed to be converted to .png or .jpg for use on the web and this was very limiting to the functionality of the final image synthesis program. The thought of including a third-party library, more specifically the semantics of linking the files correctly in the coding process, seemed more difficult than writing an original class.

4.2.2 Processing

The programming language, Processing, approaches the preliminary preparation milestone in a manner that is significantly different from that of C++. Whereas I was able to avoid IDE's for C++, doing so with Processing would have proven impossible as it happens to *be* an modified and specially engineered IDE for the Java programming language itself.

Installation of the Processing language is simple. A compressed application file is downloaded from the processing.org website. Once uncompressed an application file can be clicked and opened. This starts the Processing application and development can begin. The processing.org website provides downloads for the three major operating systems, Windows(32-bit and 64-bit), Linux(32-bit and 64-bit) and Mac OS X. Processing does not require installation so it can be run from anywhere on your computer, which means that Processing can even be saved to, and run from, an external hard drive, allowing the developer to work from any computer without hav-

ing to reinstall an IDE or compiler. The portability of Processing is convenient for students who may not have a laptop and are forced to switch between workstations.

Processing has a vector math library built into its IDE called PVector. PVector has extensive documentation on the processing.org website. Despite this, the syntax of PVector operations is anything but straightforward. This can best be described with an example. Consider the C++ equation for determining the hit point along a ray from the ray's origin:

$$p_{hit} = ray.Origin + t * ray.Direction \tag{4.1}$$

This same equation in PVector would be written:

$$p_{hit} = PVector.add(cast.Origin, PVector.mult(cast.Direction, t)) \tag{4.2}$$

Rather than overload the math components of the Vector object, each operator is expressed as a function. Thus, "+" becomes "add" and "*" becomes "mult.". This leads Processing operations to grow long and segmented, which causes confusion for the programmer. As can be seen in the above example, the C++ syntax in Equation 4.1 is much easier to read, understand, and therefore debug, than the Processing syntax in Equation 4.2.

In addition to the PVector library, Processing also has included its own image library. Unlike PVector this image library is very convenient, although no math operations were needed to be performed on image objects. The PImage library has the ability to save or open any type of common image file, as long as the file type is specified within the command.

Processing introduced another challenge when dealing with color. Color in Pro-

cessing is defined as a data type, which is saved as 32-bits of data formatted as AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB, where (A)lpha, (R)ed, (G)reen, and (B)lue components are all stored as 8 bits of data. Within a ray tracer it is necessary to have access to each individual color value in order to take color averages, since you need to calculate each color value individually. To extract each color value from the color type a process called bit shifting must occur. This can be shown as follows(example taken from processing.org):

```
color argb = color(204, 204, 51, 255);
int a = (argb >> 24) & 0xFF;
int r = (argb >> 16) & 0xFF;  // Faster way of getting red
int g = (argb >> 8) & 0xFF;   // Faster way of getting green
int b = argb & 0xFF;          // Faster way of getting blue
```

Listing 4.1: Java bit shifting to extract color data

Bit shifting is a technique that exposes the base functionality of how computer's save information within their internal storage. The process of bit shifting is not trivial and is confusing for new programmers who are unfamiliar with computer architecture. As can be seen by Figure 4.1, Processing has made available information and resources to solve some of the more complicated challenges introduced by the Java language.

4.2.3 Python

Many of the same complications that arise with C++ also occur with Python. Like C++, Python does not have a vector math library built in. It is also rather complicated to install correctly on a Windows machine.

Unlike C++, however, header files are not necessary for Python programs, nor do the programs have to be compiled before running. Python also comes pre-installed on Unix-based systems so development can begin right away on Linux and Macintosh

computers. Running a Python program can be done through the computer's terminal in much the same way as C++.

Although Python does not have a vector math library, I found it easier to find third party vector libraries written in Python. This might be to due to the surge of popularity that the Python language has experienced over the past few years. For this thesis, a vector library was written using the C++ vector library as a template. The decision to write an original Python vector library rather than using a pre-existing library was made to take advantage of an educational exercise that would help familiarize me with the Python language.

The imaging library that was used for this thesis is called the Python Imaging Library(PIL). PIL was originally a Python-supported library until their most recent release. PIL is now a third party image library that is still under development. PIL is only compatible with Python version 3.0 or earlier, so if a later version of Python is used PIL cannot be included in the project. It is recommended to avoid writing a custom image library and instead to use a version of Python that supports PIL because it is more beneficial to have the PIL capabilities rather than write a image class from scratch.

4.2.4 RenderMan©

Compared to the other three programming languages in this thesis, RenderMan© is an outlier. It does not follow the same milestones as the other coding languages which may appear to be a huge advantage, but there is a major drawback. RenderMan© is a very expensive rendering engine that is not available to the average computer graphics hobbyist. As of the writing this thesis in 2014, a RenderMan© Floating Institutional license sells for $274.00. The same is true for a Pro Server license. A one year student subscription costs $199.95 and does not allow students to

produce anything for commercial value. The yearly cost times four years of schooling equates to $800 on top of tuition and other expenses. It is not in my budget, nor in the typical students' budget, to have access to RenderMan© outside of an academic setting. Thankfully Texas A&M has an Institutional license for RenderMan© that was used for this thesis. This cost ranked RenderMan© lowest of all the languages for accessibility and portability.

It is worth noting that there is an open source alternative to RenderMan© called Pixie©. Pixie© is based off of the RenderMan© syntax but does not have all of the same features that RenderMan© does, nor was it developed by the same people at Pixar. If learning the RenderMan© syntax and RIB/RSL file structure is desired however, then Pixie© is an affordable alternative to use from home or outside of an academic environment.

4.3 Milestone 2: Direct Illumination- Ray Casting

A majority of the planning and learning involved in this research occurred on an implementation level during this second milestone because it was the first milestone to implement code. Declaring a class was a different learning experience for each language. C++ and Processing have a similar syntax but Python is different both in syntax and in theory.

A class is fundamentally a descriptor of characteristics and behaviors needed by a virtual "object" in a computer program. Class declaration and planning for each language is important because they relate back to accomplishing the ray-tracing theory. The goal, remember, is to determine which object, if any, intersects with each pixel in an image. This is done by iterating over every object in a scene to determine if it intersects with the pixel in question. The goal is to have all of the data storing each object in the scene held in one place so that only one iteration loop

needs to be written. Since it is unknown how many objects are in a scene at one time, the container size for holding the data for the objects needs to be dynamic in size. This introduces the two main challenges for implementing basic ray casting theory in a computer program: organization of classes(objects) and assembly of those classes into a dynamically-sized data container (or data structure).

4.3.1 Classes Overview

In order to successfully determine how classes should be established, it is important to plan how classes should relate to one another. The relationships of one class to another can most successfully be summarized in Figure 4.1, found in Section 4.2. The picture represents the four main object types: **shapes**, **materials**, **color**, and **lights**. In summary, every **shape** must have a **material**, which must have at least one **color**.

The different types of **shapes** are represented by the circles sprouting locally around its pink square (sphere, ellipsoid, cylinder, etc.), just as the type of **material** can be any of the circles sprouting from its pink square (Lambert, Phong, Cartoon). The circles represent child classes. Child classes can inherit the properties and behaviors of their parent class. **Lights** do not have any connection with **shapes** or **materials**, but each **light** must have a **color**, which then provides color information for a **material**. Classes can be broken into two main elements: variables, which hold data within the class, and functions, which modify the data that is held within the class. In all languages variables have a specific type, for instance integers (whole numbers), floats (decimal numbers), and strings (words or letters). Defining custom classes is a way the programmer can introduce new variable types. The following is an abstract summary of each of the primary variables and functions for each major class type established in this raytracer program. For simplicity sake we will just

discuss the functions vital to basic ray casting theory, rather than any extra helper functions that might have been implemented.

4.3.1.1 Shapes

The shape parent class consisted of four variables and three functions. The class structure is demonstrated in the Unified Modeling Language (UML) diagram, Figure 4.2, below.

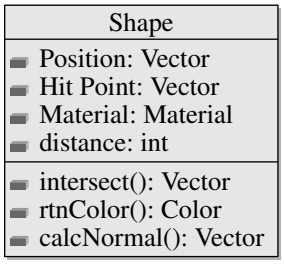

Figure 4.2: Unified modeling language (UML) diagram for the shape class

As shown above, the shape class had all the important base functionality for every shape object. Associated with every shape object was a vector object that described its position in 3D space (Position), a Vector that was calculated to describe where a light ray or camera view ray would hit the shape (Hit Point), a material object that described what the shape would look like when interacting with the light and camera view vectors (Material), and a distance integer that helped to sort all the shapes in the scene from closest to farthest away(distance). The shapes initially had three important base functions that calculated the following: if the shape intersected a light ray (intersect), what material on the object the return color would be based off of(rtnColor), and the surface normal that was given from the intersection point (calcNormal). These three functions were inherited by all of the children shape

objects.

4.3.1.2 Color

The color parent class incorporated all types of color information that could be displayed in a material. The color objects were vectors that held Red, Green, and Blue(RGB) hue information with the ability to be added and subtracted from each other. Color information could also be taken from an image or a procedural equation to describe the RGB information transmitted from the surface. Figure 4.3 shows the color class.

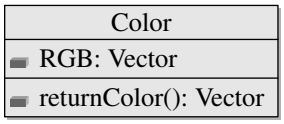

Figure 4.3: UML diagram for the color class

The color parent object held Color information in a Vector (RGB) and a method that described how to calculate the color vector(returnColor()).

4.3.1.3 Materials

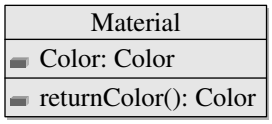

Figure 4.4: UML diagram material class

As can be seen from Figure 4.4, the material class very simply had a variable that stored a color object (Color), which described which color would be used when

calculating the returned color from the returnColor function. This variable was either overwritten or added onto in child classes, but the base material parent class was used as a holder for all material objects.

4.3.1.4 Lights

A light object was very simple and followed the UML Diagram in Figure 4.5. A parent light object held only the information needed to determine the color for

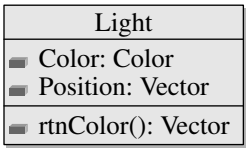

Figure 4.5: UML diagram for the light class

the light so the variables were the color object and position vector. The function calculated whether or not the light intersected with the object and if so, the color that would be returned from the light's color object.

4.3.2 Declaring Classes with each Language

For this section, we will use the Shape class to demonstrate how classes are declared in each language. The Shape class structure is defined in Figure 4.2.

4.3.2.1 C++

```
1 //  Shape Class
2 #ifndef SHAPE_H
3 #define SHAPE_H
4
5 #include "../classes/Vector.h"            //Vector class
6 #include "../materials/material.h"        //Material class
```

```
#include "../lights/light.h"                //Light class
#include "../utils/ray.h"                   //Ray class
#include <iostream>
#include <string>

using namespace std;

class Shape
{
  protected:
    Vector3d position;
    Material *mat;
  public:
    Vector3d hit_point;

    //constructor
    Shape(){
      position = Vector3d(0,0,0);
      mat = new Material();
      cast = true;
      blur = false;
    };
    Shape(Vector3d pos){
      position = pos;
      mat = new Material();
    };
    ~Shape(){};
    //function to return the Color Vector returned
    //from a light ray intersection on with the material
    virtual Vector3d matReturn(Ray ray, Light *light)
    {
        return mat->rtnColor();
    };
    //function to determine hit_point and return
    //boolean to determine if surface of object was hit.
    virtual bool intersect(Ray cast){
        //calculate if ray intersects object's position here
   ...
    };
};
#endif
```

Listing 4.2: C++ Class Example

```
1 Sphere sphere_object = new Sphere();
2 //if ray object was created we can pass the object into the
     intersect function.
3 bool is_hit = sphere_object->intersect(ray_object);
```

Listing 4.3: C++ Class Usage Example

4.3.2.2 C++ Explained

The C++ declaration of a class has a few quirks that are different from other languages. One: "include guards" (found on lines two and three of Listing 4.2). An include guard prevents the Shape.h class from being defined multiple times, which can happen if Shape.h is included, or referenced, into multiple classes within the ray tracing program, especially within classes that inherit from each other. Include statements are found on lines five through ten of Listing 4.2. Inclusion of these has the potential to cause an error and inhibit the program from running. It can be argued that every class can be written in one extremely long document and remove the need for include guards and include statements, but this will create a document that is difficult to debug and troubleshoot when errors occur. Determining all the needed include statements when creating a class for the first time can be tricky and forgetting to include necessary classes will cause an error.

Line twelve demonstrates a standard practice in C++ that describes in which context the file can be used, which in C++ is called a namespace. For the sake of simplicity, all files in this project use the standard, or std, namespace.

The declaration of the class begins on line fourteen. The keyword "class" followed by the name of the class and curly brackets signifies the start of a class. Within the brackets variables can start to be declared and and the functions necessary to the shape class. Line sixteen begins the declaration of class variables. Variables can be defined in one of three groups in C++: public, protected, or private. The protected

variables that are declared on line sixteen can only be accessed by classes that inherit from the shape class. The public variables declared on line nineteen can be accessed from any program that creates a shape object. As can be seen, we have grouped each variable and function inside of a "protected" (line 16) and a "public" grouping (line 19). For this thesis, most variables were considered private or protected, but there is no issue with leaving all variables public. Public, protected, or private variable scope needs to be considered when trying to create an object-oriented ray-tracer. Having protected variables allows for more confidence in the values of variables staying exact because access to the variables is limited to methods provided within the class.

Next in the figure is the constructor on line 23 and 29. A constructor is a required function that instantiates a class. Without a constructor the first line of Listing 5.1 would not be possible. Constructors are defined like other functions but with a name that matches the class name. Two constructors are defined as a shortcut to setting the internal variables of a class. Line 29 allows the creation of a shape class at a specified coordinate as opposed to 0.0, 0.0, 0.0. Following the constructor are the functions that are used to calculate various important information. Since the functions defined on line 36 and line 42 are grouped under the "public:" section of the class, these functions can be called as elements of the class after they have been instantiated.

To declare a C++ function the scope needs to be determined, the variable type that will be returned declared, and the function named. On line 42 of Listing 4.2 we declare a public function(because of the public grouping earlier in the code) that returns a boolean, or true/false value, in a function called "intersect". The *virtual* keyword is discussed in later sections.

All of these components need to be considered when declaring a class with C++. If any of these things are not correct a variety of errors can be occur that need to be

handled before the ray tracer will complete.

4.3.2.3 Processing

```
/*
  Abstract Shape Superclass encompassing all shape objects
*/
abstract class Shape{
  //Global Variables
  PVector hit_point;  //hit point on the surface of the object

  //Protected variables only to be used by Shape subclasses
  protected PVector position;  //position of the Shape in 3D
   space
  protected Material mat;  //Material assigned to this shape

  //Basic Shape constructor
  Shape(){
    position = new PVector(0,0,0);
  }

  //Another Shape Constructor
  Shape(PVector pos){
    position = pos;
  }

  //Another Shape Constructor
  Shape(PVector pos, PVector up){
    position = pos;
  }

  //function to return the Color Vector returned
  //from a light ray intersection on with the material
  PVector matReturn(Ray ray, Light light)
  {
    return mat.rtnColor();
  }
  //function to determine hit_point and return
  //boolean to determine if surface of object was hit.
  boolean intersect(Ray cast)
  {
    //calculate if ray intersect's objects position here...
```

```
38      return true;
39    }
40  }
```

Listing 4.4: Java Class definition example

```
1 Sphere sphere_object = new Sphere();
2 bool is_hit = sphere_object.intersect(ray_object);
```

Listing 4.5: Java Class definition example

As has been shown, Processing class declarations have some similarities to C++ classes, but also have some differences. Unlike C++, Processing does not require any include guards or include statements. Including is accomplished by the Processing IDE. All tabs that are open within the Processing "sketch" are included and available to every other file within the project. Ignoring the abstract keyword for now, line four of Listing 4.4 is the same as C++. Processing follows the constructor format of C++ as well. A difference between C++ and Processing is the variable scope declarations. In C++, variables and functions were grouped under "public:" and "protected:" tags. With Processing, however, each variable must be individually declared *not* public, which is the default scope declaration, as can be seen on lines 10 and 11.

4.3.2.4 Python

```
1 """#############################################################
2   Shape.py File
3 #############################################################"""
4 import sys, math
5 import Vector, Material, Ray
6
7 class Shape(object):
8   __slots__ = ('hit_point', 'position', 'mat')
```

```
9
10    #constructor
11    def __init__(self, position = Vector.Vector3d(0.0,0.0,0.0)):
12      self.position = position
13      self.mat = Material.Material()
14
15    #function to determine hit_point and return
16    #boolean to determine if surface of object was hit.
17    def intersect(self, cast):
18      #calculate if ray intersects object's position here...
19      return true
20
21    #function to return the Color Vector returned
22    #from a light ray intersection on with the material
23    def matReturn(self, ray, light):
24
25      return self.mat.rtnColor()
```

Listing 4.6: Example of class definition in Python

```
1 sphere = Sphere.Sphere();
2 is_hit = sphere.intersect(ray_object);
```

Listing 4.7: Python Usasge Example

Python is the most different from C++ and Processing. While the other two languages separate blocks of code using curly braces, Python uses white space to define its code structure, so function and variable scope are determined by tab depth and the colon(":") operator. This was a big hindrance in development of code because the scope of each for-loop or function needed to be perfect, and if code blocks are not highlighted by a text editor, the Python indent can get confusing and lead to code working improperly. To create a class in Python, include statements like C++ must be used, as can be seen in Listing 4.6 on line 4 and 5. Include guards, however, do not need to be used. Python's keyword for include is "import". Utilizing the imported classes is different than in C++ or in Processing. As can be seen on line

13, in order to use any methods from the imported class the function or variable name must be prefixed with the name of the import class it is being taken from. In this case, the constructor function from the imported Material class is used with Material.Material(). Declaring a class on line 7 is the same as with C++ or with Processing, the exception being that within the parentheses it needs to be specified what the class inherits from. The default is object, because all classes are objects.

One way to declare class variables within a Python class is the "__slots__" notation, which is found on line 8. This is a more advanced technique that will save space within the final implementation process, but is not always necessary. An effort was made to try to get Python classes to look as similar to their C++ and Processing counterparts within the context of the thesis.

Normally within Python, variables can be declared and added to a class from anywhere after a class has been instantiated. Slots limit the number of available variables to specific variable names, which saves memory in the long run but also prevent extra or unwanted variable declarations within instantiated objects. The concept of variable scope in Python is foreign. To accomplish protected or private variables in classes, a convention called "name mangling" is followed, which simply adds at least one underscore before a variable name. This, however, does not cause the variable name to act protected or private; it just makes it harder to guess what the variable name is when trying to access it from another application that is instantiating a python object. For this thesis, the Python ray caster used only public variables.

The method for declaring a constructor in Python is also different. The constructor syntax for Python can be seen on line 11. Python easily allows for declaration of a constructor. Unlike C++ and Processing, all Python functions require the "def" keyword to establish them as functions. Whereas in C++ and Processing a function

with the same name as the class declares a constructor, the distributed keyword __init__ function is the constructor argument for the Python class. Classes are still instantiated with a function that mimics the class name, however, as is seen in Listing 4.7, line 1. The "__init__" function will be run when a function that mimics the class name is called. As in every Python function, the first argument provided to the function must be "self", which allows the Python function access to its own information. The self keyword is demonstrated in line 12 within the constructor where the position variable passed into the function is set to be self.position, which refers to the position variable established within the __slots__ array. Without a reference to "self", the constructor would have no access to any of the class variables. The self keyword is Python's way of handling scope. The second argument, "position = Vector.Vector3d (0.0 ,0.0 ,0.0)", eliminates the need for a second constructor because this sets a default for the position argument of the constructor if no value is provided when the constructor is instantiated. This accomplishes the same thing that was done in Listing 4.2 line 23 and line 29.

4.3.3 Dynamic Class Variables

Since Processing and C++ are both static-typed languages, a unique challenge occurs. Line 18 of Figure 4.2 and line 10 of Figure 4.4 shows that available of type material has been declared. This means that only a class that is instantiated as a material object type can be saved as that object's variable. Theoretically, however, a shape class should be able to hold any of an infinite range of object types in its mat variable. Even though this variable should have the same properties as the Material class, it should not have to be restricted to being a Material-type object. instead, it should be allowed to be a Lambert object, a Phong object, or even a Cartoon object. If an attempt was made to create a Phong object and save it within a shape object

as the mat variable, this would not work and cause an error upon compilation.

To give some insight as to why each language was designed this way, Cardelli and Peter describe strong-typed, or static-typed, languages in their article entitled *On Understanding Types, Data Abstraction, and Polymorphism* as follows:

> Static typing allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type-consistent. It facilitates early detection of type errors and allows greater execution-time efficiency. It enforces a programming discipline on the programmer that makes programs more structured and easier to read. [4]

Eliminating this data type issue is not possible but there is a solution to saving a Phong object as a material variable. Two computer science terms called *inheritance* and *polymorphism* help to solve this problem. Inheritance is the idea that a child class can inherit the properties and functions of a parent class. Making a Phong object a child of a Material object, enables the Phong object to be saved as a material variable within the shape class without throwing an error. This is accomplished in C++ as follows:

```
//material.h file//
class Material
{
    public:
        Material(){};   //constructor
        Color color;
        virtual float returnColor(){
            //calculate material color for shape hit point
        }
}
```

```
//phong.h file//
```

```
#include "phong.h"
class Phong: public Material
{
    public:
        Phong(){};  //constructor
        Color color2;   //new color variable specific to phong
    class.
        float returnColor(){
            //calculate sphere size logic.
        }
}
```

```
//lambert.h file//
#include "lambert.h"
class Lambert: public Material
{
    public:
        Lambert(){};
        Color color2;
}
```

Listing 4.8: C++ Inheritance Example

In the above example, all classes have at least one color variable and a returnColor() function because of inheritance. The Lambert/Phong classes each have two variables because they establish another color variable (color2) within themselves and inherit the color variable from the parent Material class. The returnColor() function for the material and lambert classes will behave the same but the Phong's returnColor() function has different logic because the logic has been overwritten(for this thesis the lambert class also overwrites this function).

All children classes need to call their own constructors in order to be instantiated. It is important to note that a child class must contain "#include 'material.h'" or it will not work correctly. In order to rewrite a parent function in a child class the keyword "virtual" needs to be used as seen on the returnColor() function(line 9) in

Figure 4.3.3. To accomplish this same procedure in Processing, classes need to be established as follows:

```
//material.pde file//
abstract class Material
{
    Material(){};  //Constructor
    protected Color color;
    float returnColor(){
        //calculate size for all object types;
    }
}
```

```
//phong.pde file//
class Phong extends Material
{
    Phong(){};
    protected Color color2;
    float returnColor(){
        //calculate size specific for sphere
    {
}
```

Listing 4.9: Java Abstract Class Example

The syntax for the cube.pde file is the same as the sphere.pde with the "extends" keyword showing that sphere is the child of the Shape class.

Inheritance only achieves half the goal. If the above Phong and Material classes were implemented and then instantiated and finally saved as a phong object type within the material variable within the shape object, all variables and functions written specifically for the phong class will be discarded and replaced by the functionality of the parent class. We will be virtually shaving off the corners of the square "phong" peg to fit it back into the circle "material" hole that has been prepared for it within shape objects storage in memory. This can be avoided with polymorphism.

Cardelli and Wegner explain different types of polymorphism, from which the type used for this thesis is called subtyping.

> Subtyping is an instance of *inclusion polymorphism*. The idea of a type being a subtype of another type is useful not only for subranges of ordered types such as integers, but also for more complex structures such as a type representing Toyotas which is a subtype of a more general type such as Vehicles. Every object of a subtype can be used in a supertype context in the sense that every Toyota is a vehicle and can be operated on by all operations that are applicable to vehicles. [4]

To introduce how this is achieved in each language, the topic of "pass-by-value" and "pass-by-reference" must be discussed. In C++, when declaring an object, two variations can be done in relation to the memory organization within the computer. The first, "pass-by-value", is the most common and every time a variable changes or moves a *copy* of the variable is made within the backend and saved in a new location. However another schema, "pass-by-reference" exists, however, which creates a reference or "pointer" back to the original variable rather than copying the data.

Subtyping will allow us to "pass-by-reference" our phong objects into a material variable and still retain the phong-specific information. Subtyping, and how it is used, is also relevant to our second main challenge for implementing basic ray casting theory: assembly of classes into a dynamically-sized data container (or data structure).

4.3.4 Data Structures

It is more difficult to accomplish the data structure task in C++ and Processing (Java) than in Python. Just as a variable must be of the same type to save it to a class's variable, all objects must be of the same type to add them to the same data

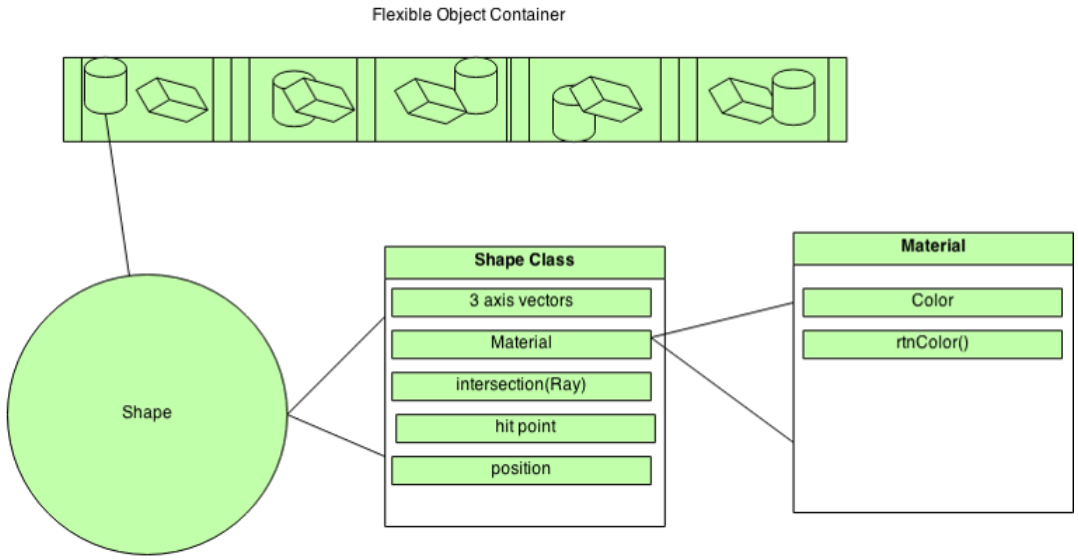

Figure 4.6: Data structure design for each program

structure. Furthermore, the most basic data structure, an Array, must be declared a specific size upon initialization. The size issue can be easily circumvented, however. For each language there is a dynamic container that will grow and shrink in size. C++ has a vector data structure and Processing has an ArrayList. To declare each is simple:

```
vector <Sphere> sphere_list;
Sphere sphere_object = new Sphere();
sphere_list.push_back(sphere_object);
```

Listing 4.10: C++ Vector Example

```
ArrayList<Sphere> sphere_list = new ArrayList<Sphere>();
Sphere sphere_object = new Sphere();
```

```
3 sphere_list.add(sphere_object);
```

Listing 4.11: Java ArrayList Example

Each of the above examples creates a *Sphere* list, an instance of a *Sphere* object, and subsequently puts that *Sphere* object into the *Sphere* list. Note that each structure *must* be declared as the type of object that is being placed into it. The "pass-by-reference" tactic in C++ is what allows different variable types to be saved to a common parent data structure, as follows:

```
1 vector <Shape*> shape_list;
2 Shape *sphere_object = new Sphere();
3 Shape *cube_object = new Cube();
4 shape_list->push_back(sphere_object);
5 shape_list->push_back(cube_object);
```

Listing 4.12: C++ Vector Example

Declaring the vector with the "*" denotes a C++ pointer, or that the variable will use the "pass-by-reference" schema. Pass-by-reference and pass-by-value variables cannot be combined within a data structure. To then call any functions within the pointer variables, "->" must be used rather than ".". This allows for all shape objects to be saved within the same data structure.

To accomplish this in Processing the Shape class was declared as abstract in Figure 4.4. This keyword signifies that an object of this type may not be instantiated within a program. This keyword demonstrates true inheritance because it acts as a placeholder for commonly used functions and variables between class types. In Java, it allows child classes to be declared as common types without losing each class's

individual functions and variables. This is accomplished as follows:

```
ArrayList<Shape> shape_list = new ArrayList<Shape>();
Shape sphere_object = new Sphere();
Shape cube_object = new Cube();
shape_list.add(sphere_object);
shape_list.add(cube_object);
```

Listing 4.13: Java ArrayList Example

Notice that sphere_object and cube_object are declared as a type shape but using the child class's individual constructors. This allows "shape_list.add()" to be called on the next two lines without throwing an error. If the Shape class was not declared as abstract, when declaring Shape sphere_object and calling the child constructor, and when calling functions that are specific to the child class, those calls will be replaced with calls to the parent class's functions.

While Java still has "pass-by-value" and "pass-by-reference", Processing/Java has simplified their language to effectively make all declared variables function as pointers, pointing back to the original variable declaration.

4.3.5 Python Inheritance and Data Structures

While parent classes are not important for Python list functionality because Python is considered a dynamically-typed language (as opposed to the static-type for C++ and Java), for organizational purposes they were used and inheritance was declared as follows (note tabs and spacing), using the same example:

```
//shape.py file//
class Shape(object):
    __init__(self,...):
        //shape constructor
    def size(self,...):
```

```
        //size calculation for all child classes
```

```
//sphere.py file//
import Shape
class Sphere(Shape.Shape):
    def __init__(self,...):
        //sphere constructor contents
    def size():
        //calculate size specific for sphere
```

Listing 4.14: Python Class Inheritance Example

4.3.5.1 Data Structure

Accomplishing the data structure task in Python is the most simple and takes no extra effort. Simply declare an object and place it in a Python list, like so:

```
sphere = Sphere.Sphere();
cube = Cube.Cube();
objects =  [];
objects.append(sphere);
objects.append(cube);
```

Listing 4.15: Python List Example

The above code creates a sphere object, cube object, and an objects list, then adds the sphere and cube into the data structure with the lists append method. Python lists can accept any combination of different variable types, which can be useful for a ray-tracer's implementation, but dangerous for software robustness. If the assumption is that all variables in a list have the same type and therefore the same associated functions, when an object that does not fall under these assumptions gets

added to the list the program will fail. More care needs to be taken to ensure all variables are of the correct type to be certain that the program will run successfully, since this is not required for the program to initially start running.

4.3.6 Milestone Results: Processing Time Overview and Images

Once the issues with class and object declaration were solved implementing the Basic Ray Casting Milestone was very straightforward with each language. There was one issue with the Processing language, however, that accounted for a majority of the Processing implementation and accounts for a core theory of ray tracing implementation.

4.3.6.1 Left Hand vs. Right Hand Rule

In the 3-D virtual world there is a set of governing rules that determine an object's position within the space, known as a coordinate plane. As in a two dimensional graph, a 3D scene contains an X and a Y coordinate, but a new coordinate, Z, is added. The order and direction for each of these axes determines many important calculations when determining size and space of objects in the scene. Left-handed and right-handed can be described as follows(from processing.org):

> "In order to draw something at a point in three dimensions the coordinates are specified in the order you would expect: x, y, z. Cartesian 3D systems are often described as "left-handed" or "right-handed." If you point your index finger in the positive Y direction (up) and your thumb in the positive X direction (to the right), the rest of your fingers will point towards the positive Z direction. It's left-handed if you use your left hand and do the same."

For the C++ and Python implementations of the ray tracers a coordinate system needed to be created from scratch. I am right handed and therefore a right-handed coordinate system was used. This means, when looking at a computer monitor, the positive Y axis points towards the top of the screen, the positive X axis points to the right of the screen and the positive Z axis points directly out of the screen, perpendicular to the screen's surface. This is not a typically traditional schema for coordinate planes with computers, nor is it the scheme used by Processing. Processing used a left-handed coordinate system where the positive y axis points towards the bottom of the screen, the positive X axis points towards the right of the screen, and the negative Z points away from the user, towards the back of the computer, perpendicular with the computer screen. Many of the calculations associated with cross products and other vector math needed the Y value and Z value negated so that the calculations behaved more similarly to the other two ray tracing programs.

4.3.7 Basic Ray Casting Images

Here are a few images from the Basic Ray Casting Milestone. While these are just a few examples from each language and each task, each language accomplished the same types of images that are presented in Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10 and Figure 4.11.

4.3.8 Performance Results

For each test in Basic Ray Casting a variety of levels of complexity were tested in order to see how each language handles a different amount of load. For each rendering a 1280 x 720 image size was used to generate a high quality image for testing. The results are graphed and labeled in Figure 4.12, Figure 4.13, Figure 4.14 and Figure 4.15.

4.4 Milestone 3: Ray Tracing and Distributed Ray Tracing

The main complications discovered for this milestone where all related to the implementation of a recursive ray tracing strategy. All three languages had the same difficulties that were not unique to each implementation. To implement recursion requires the knowledge of establishing a process function within each raytracing function that will repeatedly call itself unless specific escape conditions are met. It is important to establish these conditions such that they are met within the execution of the program or else the ray tracer will implement an infinite loop that will consume computing resources until the computer crashes and needs to be restarted.

The main challenge introduced for distributed ray tracing was the idea of randomness. In order to get a jitter, as described on page 15 for every vector value for some aspect of the ray tracing process, whether it be the hit point or the view point of the camera. The process used to achieve this effect was different in each language, but Processing and Python were surprisingly similar.

```
float randNum(float LO, float HI)
{
    float U2 = LO + (float)rand()/((float)RAND_MAX/(HI-LO));
    return U2;
}
```

Listing 4.16: C++ Random Function

```
random(0.0,1.0);
```

Listing 4.17: Processing Random Function

```
random.uniform(0.0,1.0);
```

Listing 4.18: Python Random Function

Processing and Python each had their own random libraries that handled generating random numbers between two specified values but in C++ it was not provided. Writing a random function took a lot of overhead and wasted time comprehending how computers generate random numbers in C++ at a foundational level. It was significantly easier and more intuitive to implement a random function in both Python and Processing than it was in C++.

4.4.1 Ray Tracing and Distributed Ray Tracing Images

Here are a few images from the ray tracing and distributed ray tracing milestone. While these are just a few examples from each language and each task, each language accomplished the same types of images that are presented in Figure 4.16, Figure 4.17, Figure 4.18, and Figure 4.19.

4.4.2 Performance Results

Performance results each language can be seen in Figure 4.20

4.5 Milestone 4: Indirect Illumination

Milestone 4 experienced the same complications as Milestone 3, with the majority of the implementation being the same. No new computer science concepts were introduced for this section, so this Milestone was completely based upon ray tracing theory rather than language implementation. Though this task as only accomplished by c++, by this time I had fully mastered the language and felt comfortable accomplishing the task.

4.5.1 Indirect Illumination Images

Indirect Illumination Milestone image can be found in Figure 4.21.

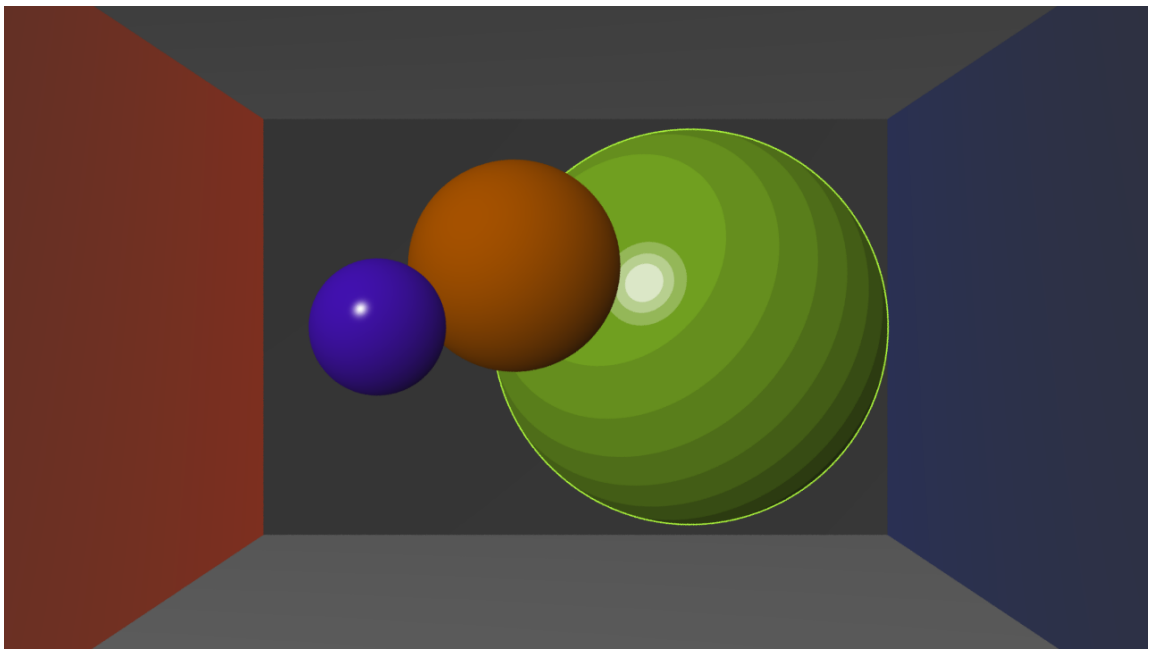

Figure 4.7: Ray casting accomplished with Processing. Shown are three spheres with a different texture type for each, and five planes all with flat textures applied.

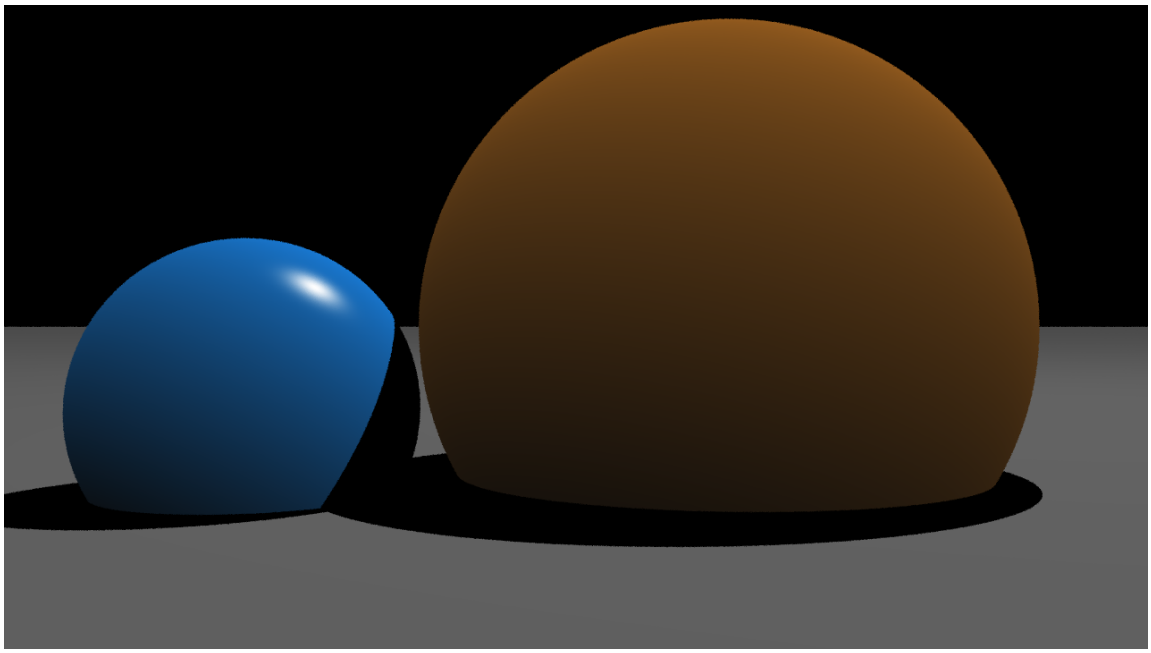

Figure 4.8: Ray casting accomplished with Python. Shown are two spheres, and one plane that all cast shadows from a simple point light.

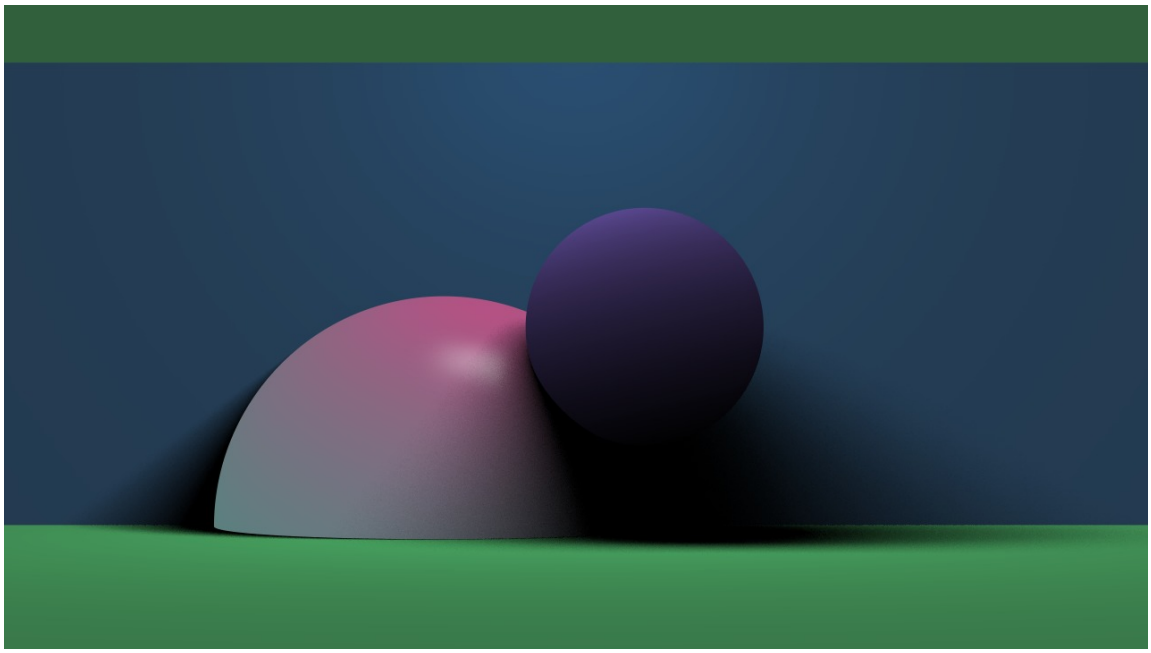

Figure 4.9: Ray casting accomplished with C++. Shown are two spheres, three planes and an area light that all cast shadows, which accounts for the soft shadow effect.

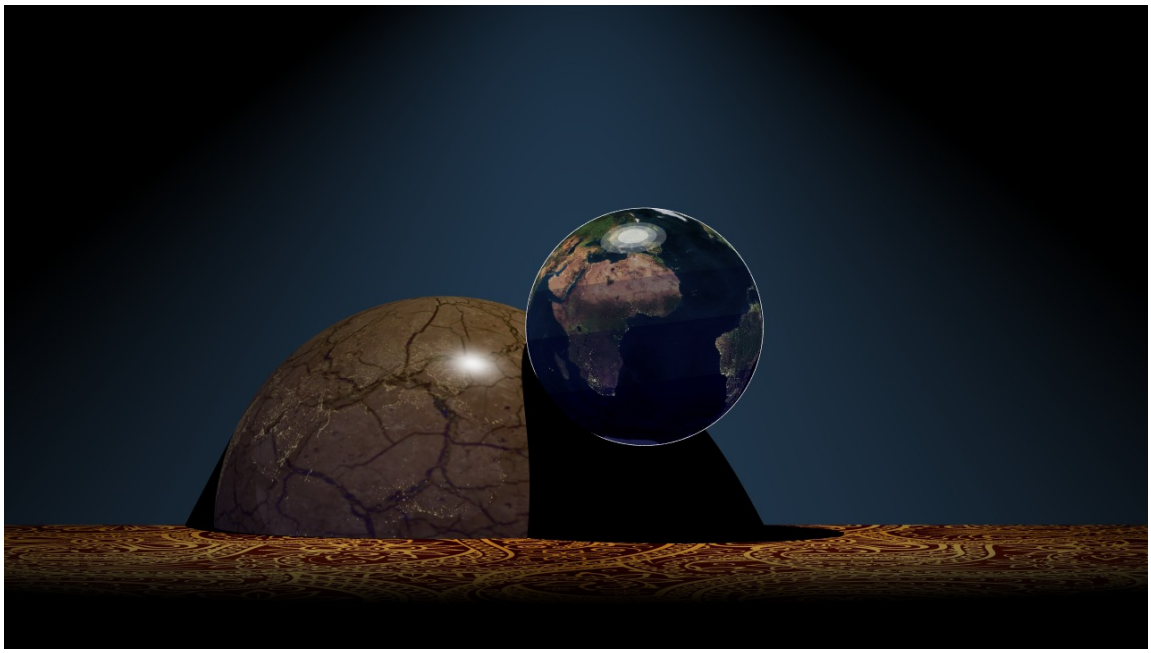

Figure 4.10: Ray casting accomplished with C++. Shown are two spheres with 2 separate shader types with two separate images mapped to them, and two planes, one with a repeated image texture and one without any image texture.

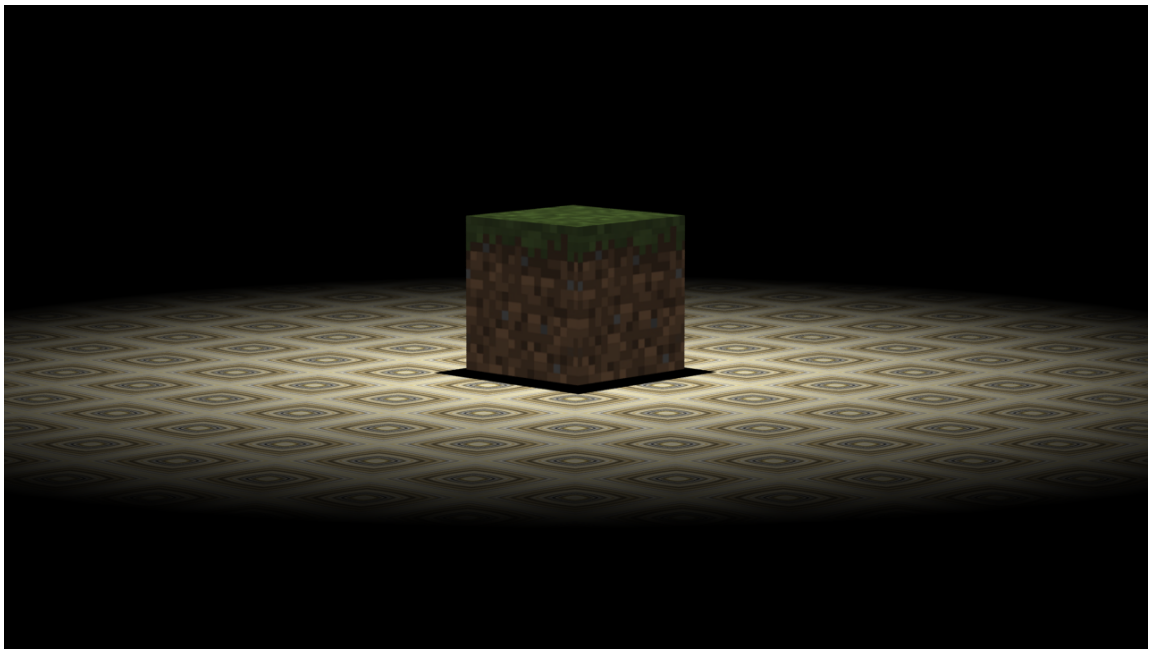

Figure 4.11: Ray casting accomplished with Processing. One cube with twelve triangles and an image mapped to it on a plane with a repeated image texture illuminated by a spotlight.

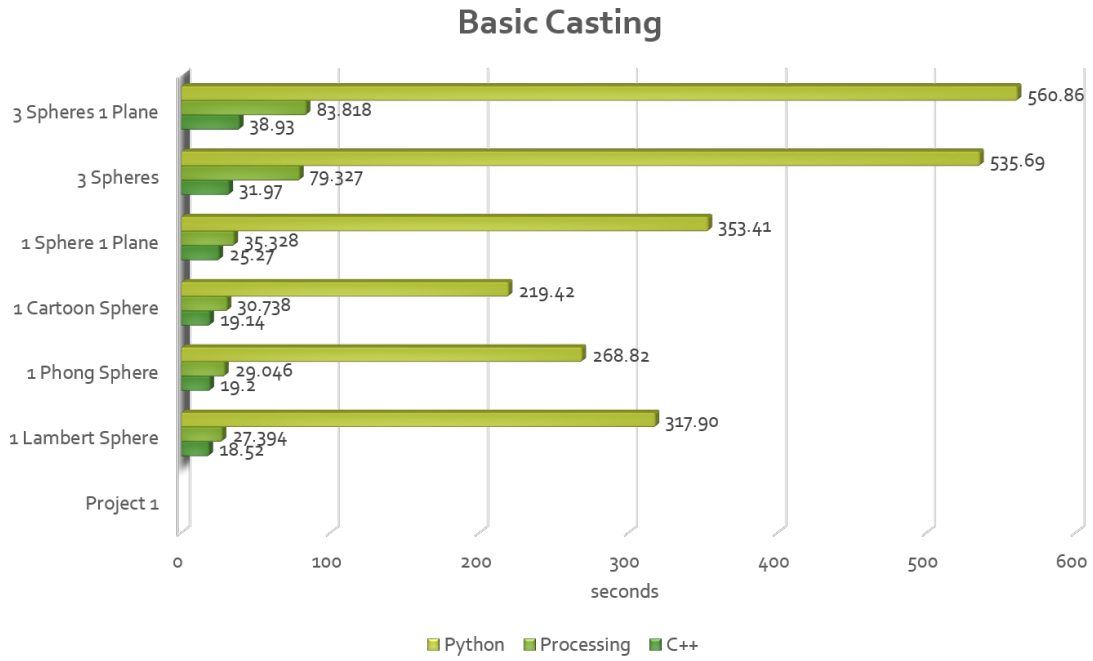

Figure 4.12: Basic ray tracing performance graph for each language

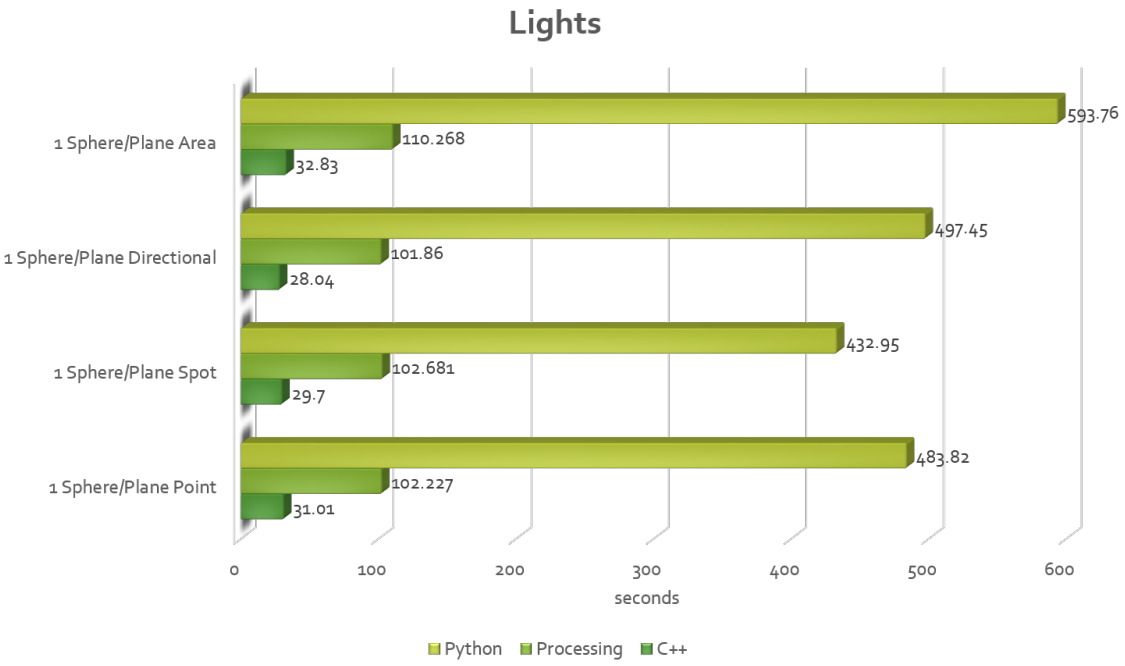

Figure 4.13: Light's performance graph for each language

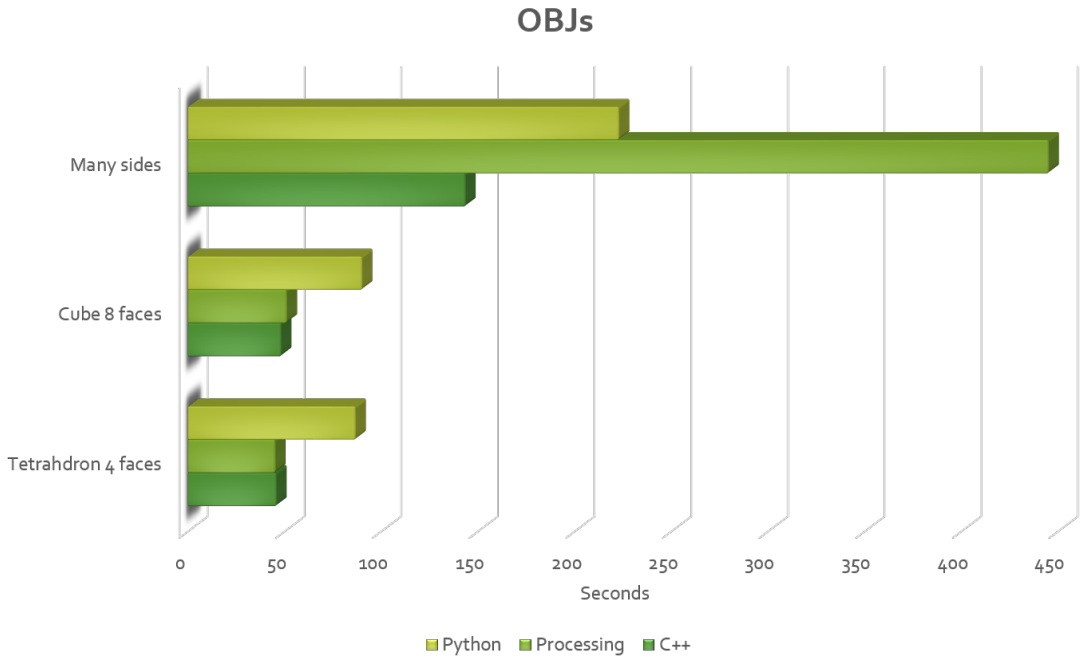

Figure 4.14: OBJ performance graph for each language

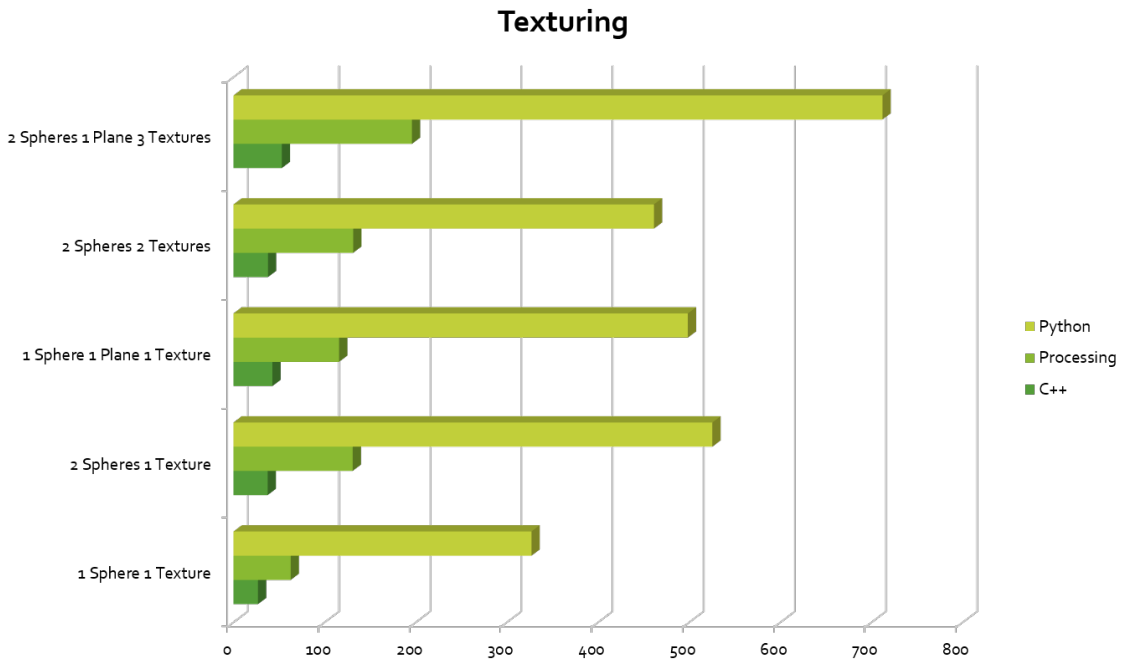

Figure 4.15: Texturing performance graph for each language

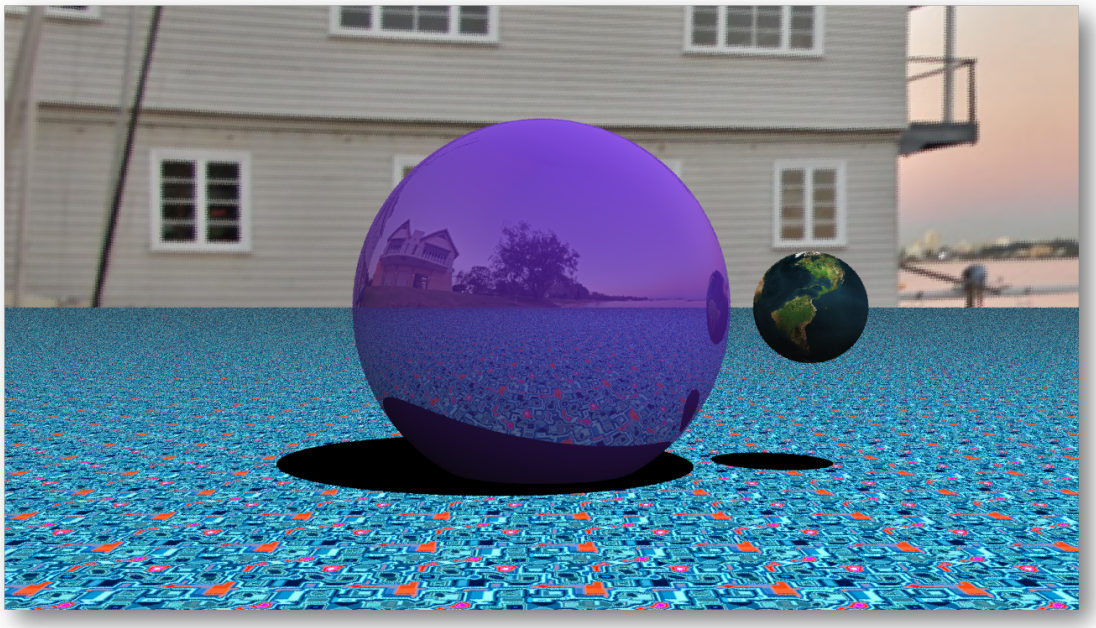

Figure 4.16: Reflection effect completed in Python

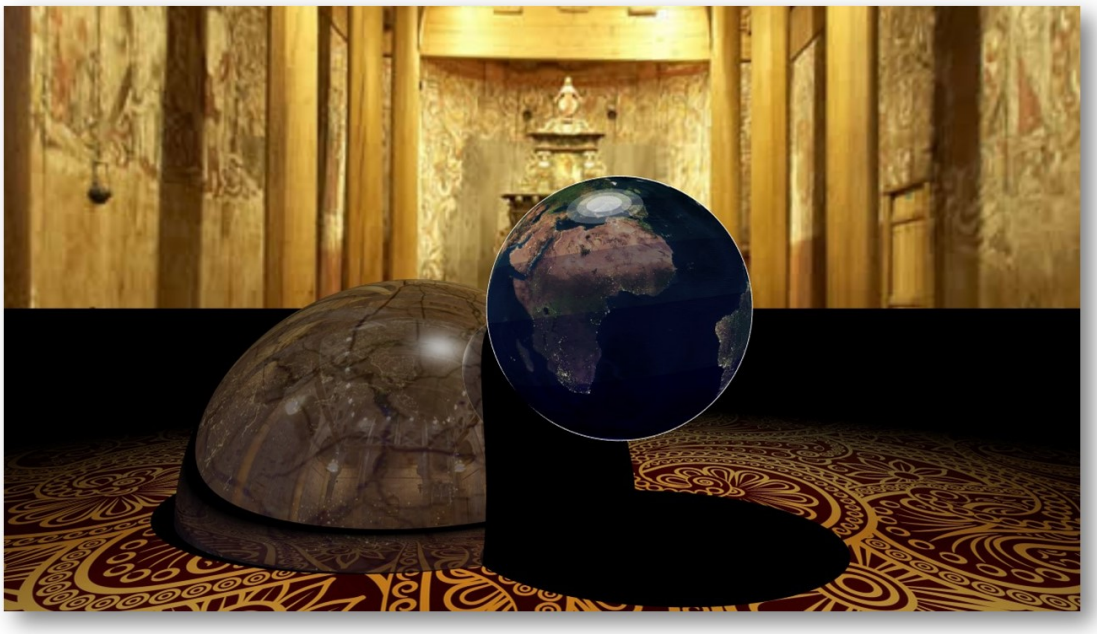

Figure 4.17: Reflection effect completed with C++

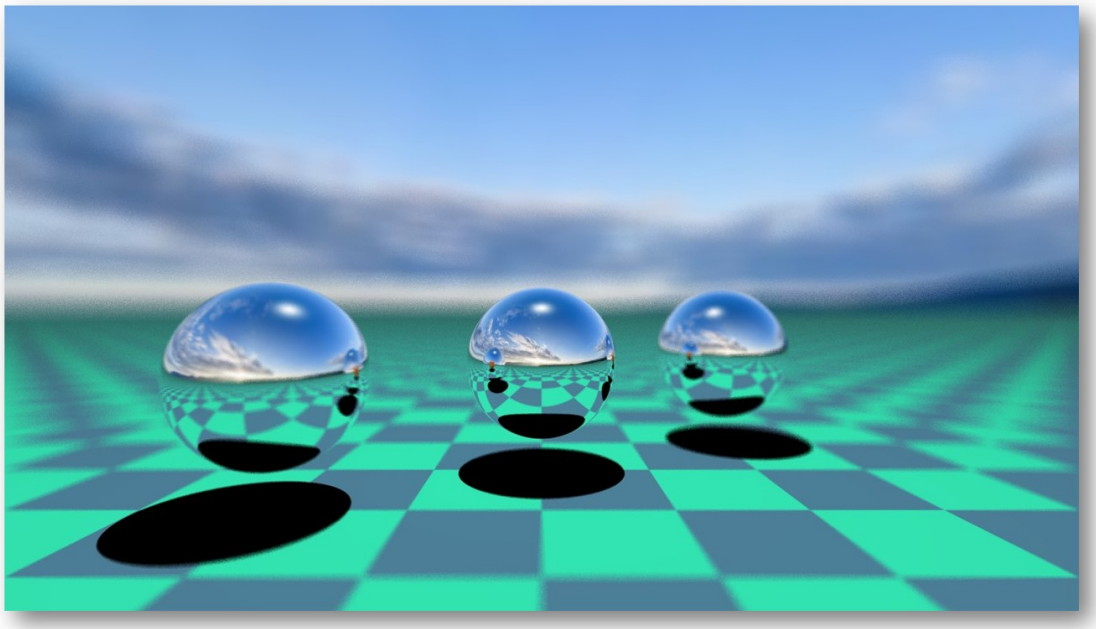

Figure 4.18: Depth of field effect captured with C++

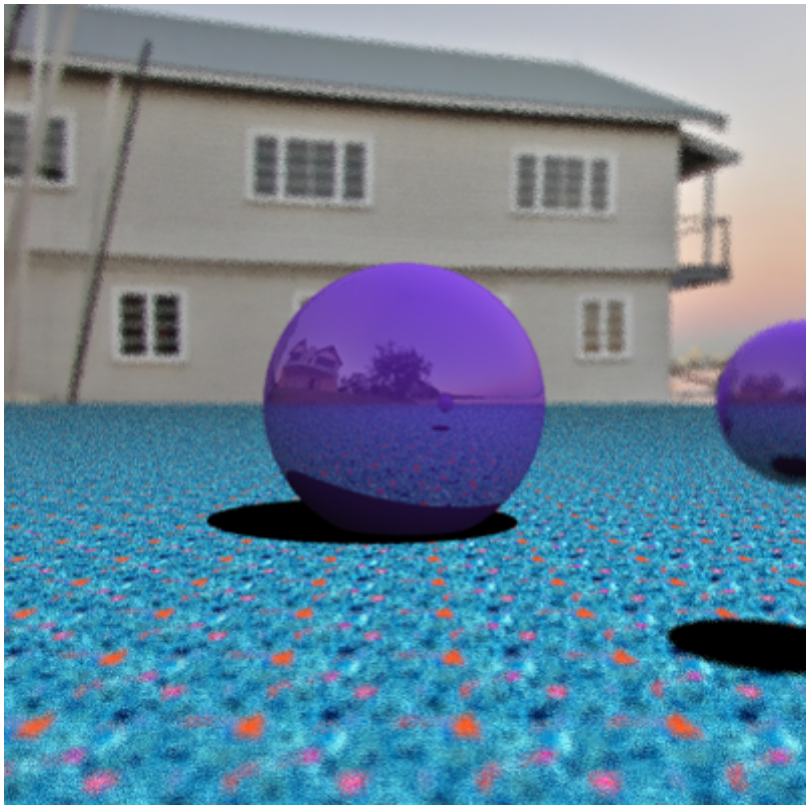

Figure 4.19: Depth of field effect captured with Python. Pixelated image shows one of the restrictions of Python computing power for an image of this scale.

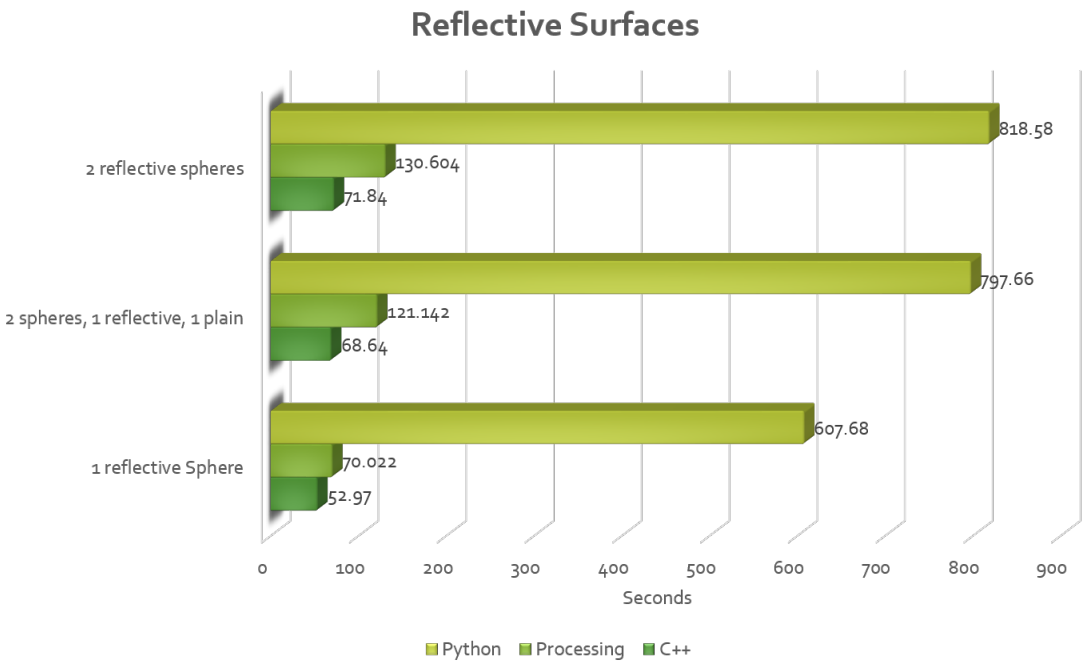

Figure 4.20: Texturing performance graph for each language

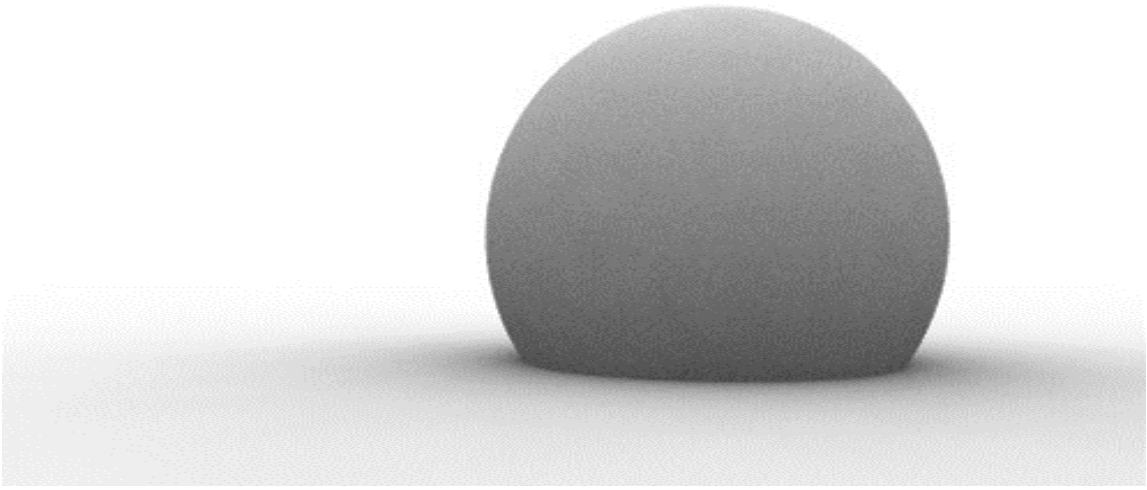

Figure 4.21: Ambient occlusion effect created with C++

5. RESULTS AND CONCLUSIONS

Each language had its advantages and disadvantages to the ray tracing process. These were further accentuated by the speed tests conducted for each Milestone. The results of each speed test were not surprising based on previous knowledge of each language's strengths and weaknesses.

Python was the slowest and C++ was the fastest; Processing was faster than Python yet still held a significant difference from C++ in large computing milestones. The speed difference between Processing and C++ is less than expected, since C++ is compiled before running the test and Processing's speed test is considered with the Processing "play" button, which presumably compiles and runs the code all at once.

5.1 Conclusions

Each language will be discussed with its pros and cons. A recommendation as to the most suitable language for maximizing focus on the implementation theory as opposed to the implementation process of writing code syntax will also be made. It is essential however, it is essential to learn a baseline of scripting and coding concepts; the language syntax and compilation processes cannot be avoided. For instance, in order to call a function associated with a class, the class must first be initiated. After which, call the function as part of that class, as follows:

```
1 Sphere sphere_object = new Sphere();
2 bool is_hit = sphere_object.intersect(ray_object);
```

Listing 5.1: C++ Class usage example

I expected some programming concepts to be easier in some languages. For instance computing procedural images from recursive math equations was very difficult for me to grasp within C++. Implementing this was no easier in Processing or Python, emphasizing that as natural or user-friendly a language's syntax might claim to be, if the underlying concepts are not fully understood, it makes no difference.

5.1.1 C++

C++ is the fastest processor of all the languages tested. Its speed allowed for quicker iterations and more creative freedom with the final images. C++ processed the scale of computation for large images far more quickly than either Java or Python. On the downside, C++ is not an easy language to learn; there was much more overhead to learn in order to compile and set up a C++ project. Whereas other languages allow for the simplicity of simply creating an executable file and running, C++ had a variety of compilers to choose from, followed by a multitude of files and commands needing to be run before the image would appear on the screen. This could be automated to save time, but the initial process is not intuitive enough to a new programmer. I assert that the knowledge gained from writing the first ray tracer in C++ allowed (or enabled faster) for faster implementation of the subsequent ray tracers. Any artist who would like a complete understanding of how programming works from the bare-bones of computer design should challenge themselves and implement in C++. Since C++ is a low-level language which exposed the basic process of how a computer handles the execution of a program with it's "make" functionality, which helped me to understand how the computer was processing and failing on different part of the process. That being said, most artists will be writing only one ray tracer, so the knowledge of how to compile and run the program still needs to be gained, but this can be accomplished with languages that are more intuitive and forgiving

than with C++.

5.1.2 Processing

Processing was the easiest language to compile and execute code. Processing has expedited the execution process by packaging the code compiler within its own custom "IDE" (The language Processing for this thesis encompasses both the java language and the Processing executable that is delivered with a download). While Processing comes with all the libraries necessary to implement a ray tracer, using them introduces another challenge. Addition in this language with the supplied Vector form was done through a function(Vector.add()) rather than through an operator (such as plus "+" or minus "-"). While unnatural for complex math functions, overall, this was a small setback in creating a raytracer. The benefits of an expedited compilation process outweighs the initial difficulty of navigating language syntax. In addition, error handling in Processing is more informative, and will bring directly you to the section of the code where the error occurred within the IDE. This is a benefit of using the Processing IDE as a part of the Processing language, over Python or C++.

5.1.3 Python

Python was the slowest of all the languages. It was, however, the easiest to write code in. By design, Python strives to have the most instinctive syntax of all the languages. While there are still rules to follow when creating Python code, Python contains far less than C++ or in Processing. Python, however, still needed the use of include statements, which can be a difficult concept to grasp when writing large programs. Since Java and C++ are both statically typed, they need more strict rules as to when variables can be set and what those variables can be set to. A lot of time was spent learning and understanding polymorphism and sub-typing for Java and

C++ so that each ray tracer can more closely mimic the ray tracing theory. With Python, this time-consuming research was unnecessary when implementing these complex data structures.

Python was significantly slower than both C++ and Processing. While Pixar or Dreamworks will not be using Python for the next iteration of their ray tracer,the slowness was inconsequential for learning ray tracing theory. Speed being its biggest impediment, strategies could be implemented to allow for a speedier artistic development process. The most useful time-saving work-flow strategy involved developing a lower resolution "layout" image which was used to prepare a final render, as opposed to rendering the full resolution for every iteration. For single frame rendering, five minutes is not a long time to wait for the final product, but when iterating to get the perfect composition, 5 minutes a frame can add up over 50-100 iterations. To iterate over artistic versions of a rendered image, a lower resolution image that only takes 30 seconds preferable. The final image only needs to be the final resolution, which theoretically should only take five minutes once.

5.1.4 RenderMan©

RenderMan© was the outlier of the group. After attempting the first milestone with RenderMan© I decided it was not a good solution to learning the basic functionality of the Image Synthesis process and it was abandoned. I was unable to replicate examples given by the RenderMan© website to learn how to develop with RenderMan©, and the overall documentation for the core RenderMan© scripting functionality is sparse. In addition, the images produced by RenderMan© had too many other variables involved with the render (i.e. indirect illumination factors that worked into the equations), so using it as a learning tool to focus on specific aspects related to Milestones established in this task was impossible. The overall experience

working with RenderMan©'s scripting functionality was more frustrating than educational and is determined to disadvantageous to learning image synthesis theory. The final recommendation is given with no consideration of RenderMan© .

5.2 Final Recommendation

For every language, you need to learn computer science terms like classes and recursion. This is beneficial to the overall process and cannot be avoided. In addition, it is understood that in each language there is a necessary evil of learning some sort of syntax-related information. Every language has its own quirks, and these quirks establish the difficulty level in using each language. Time needed to learn each language to process syntax and data types was relatively equal for each language. Therefore, the deciding factor that this recommendation is based on is the setup overheard that is required to learn before any code is written.

Processing is the best recommendation considering it's limited-to-no overhead. While Processing has some counterintuitive syntax, learning that syntax takes no more time than learning the syntax of C++ or Python, and and takes less time to learn than the total time it took to learn how to set up the Python and C++ environments. With C++ you must understand make files and header files in addition to establishing a workflow to successfully execute the written code, but Processing has packaged all this overhead into a simple IDE that "makes" and executes autonomously. Processing's compilation times were not as slow as Python, and compared to C++ the times do not discredit the language from quick, interactive artistic repetition to achieve the best results. Processing was initially created "to teach computer programming fundamentals within a visual context" and after testing the different languages, Processing was the language with the least overhead to implementation benefit ratio and is highly recommended for artists wishing to

introduce themselves to complex programming problems.

REFERENCES

[1] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[2] James F Blinn. Simulation of wrinkled surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 12, pages 286–292. ACM, 1978.

[3] James F Blinn and Martin E Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[5] Edwin Catmull. A subdivision algorithm for computer display of curved surfaces. Technical report, DTIC Document, 1974.

[6] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452. ACM, 1998.

[7] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984.

[8] Henri Gouraud. Continuous shading of curved surfaces. *Computers, IEEE Transactions on*, 100(6):623–629, 1971.

[9] Henrik. Ray trace diagram. https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg.

[10] Cay S. Horstmann and Gary Cornell. Core java 2: Volume i–fundamentals. pages 3–18, 2002.

[11] John F. Hughes, Andries van Dam, Steven K. Feiner, Morgan McGuire, and David F. Sklar. *Computer graphics: principles and practice*. Pearson Education, 2013.

[12] Kri. Gouraud low. https://commons.wikimedia.org/wiki/File:Gouraud_low.gif.

[13] Richard F Lyon. Phong shading reformulation for hardware renderer simplification. *Apple Computer*, 43:6, 1993.

[14] Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420. ACM Press/Addison-Wesley Publishing Co., 1997.

[15] Steve McConnell. *Code complete*. O'Reilly Media, Inc., 2004.

[16] David R. Naugler. C# 2.0 for c++ and java programmer: conference workshop. *Journal of Computing Sciences in Colleges*, 22(5), 2007.

[17] Addy Ngan, Frédo Durand, and Wojciech Matusik. Experimental validation of analytical brdf models. In *ACM SIGGRAPH 2004 Sketches*, page 90. ACM, 2004.

[18] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[19] Scott D. Roth. Ray casting for modeling solids. *Computer graphics and image processing*, 18(2):109–144, 1982.

[20] Bjarne Stroustrup. A history of c++: 1979–1991. In *History of programming languages—II*, pages 699–769. ACM, 1996.

[21] Guido van Rossum. Personal history- part 1, cwi. http://python-history.blogspot.com/2009/01/personal-history-part-1-cwi.html, 2009.